UNIVERSITAT DE BARCELONA

UNIVERSITAT ROVIRA i VIRGILI

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

FACULTAT D'INFORMÀTICA DE BARCELONA (FIB)
FACULTAT DE MATEMÀTIQUES (UB)
ESCOLA TÈCNICA SUPERIOR D'ENGINYERIA (URV)

Adapting the Smart Python Agent Development Environment for Parallel Computing

Master Thesis

Dmitry Gnatyshak

Supervisors:
Javier Vázquez Salceda, Department of Computer Science, UPC
Dario Garcia Gasulla, Barcelona Supercomputing Center

Barcelona, 15 April 2019

Abstract

Agent simulation models are one of the proven solutions for analysing complex problems in cases where exact modeling is either impossible or time-consuming. By defining various actors in the simulation we can get an overview of possible emergent behaviours and get some valuable insights into the problem.

The most advanced agent-based models make use of complex reasoning models that allow to emulate complicated behavioural patterns. However, all of the most sophisticated reasoning approaches typically used to enable advanced multi-agent behaviors have high time complexity, making them inapplicable for massive simulations.

On the other hand various HPC frameworks have been developed as a powerful tool for alleviating high computational demands of different models. Unfortunately, none of these integrate advanced (and thus, computationally expensive) reasoning methods.

In this work we propose a model and an implementation of a system that provides an agent simulation framework on an HPC environment, while enabling complex reasoning methods.

We formalize the model of the system, discuss the implementation aspects, and provide performance evaluation results. Finally, we showcase the real-world-based simulation example, detailing its model, implementation, and test results.

Acknowledgements

First of all I would like to express my gratitude to my advisors, Javier Vázquez Salceda and Dario Garcia Gasulla. Their guidance and insights helped me to shape this project and turn it into a useful and purposeful simulation tool.

Also, I would like to thank Sergio Álvarez Napagao and Luis Oliva Felipe for their contribution to discussions on the agent model and the river basin scenario. They helped to enrich the project and make it fit for full-fledged real-world applications.

My sincere thanks go to Julian Padget for his invaluable comments on the model and discussions on the system formalization.

I would like to thanks Prof. Ulises Cortés for starting this project and making it possible for us to come up with a final product that allowed a number of advanced frameworks to work together for a common purpose.

I also express my extreme gratitude to COMPSs, dataClay, and BSC support teams for their timely and sophisticated help in rooting out the problems I faced along the way.

Additionally I would like to thank Barcelona Supercomputing Center for providing the access to their HPC clusters for thorough testing of the system.

I would like to also thank the SPADE team from Universitat Politècnica de València for the early discussions on integration of our framework with this agent platform. Though the actual use of SPADE in the project did not work out, I am grateful for their initial contributions and hope for possible collaborations in future.

Finally, I would like to thank my family, friends, and my partner, for they have always supported and encouraged me on my path.

Contents

Chapter 1

Introduction

Agent-based modelling is a computational approach that allows abstracting complex environments where the sensing, the decision making and/or the actuation are distributed across the space or among several stakeholders. According to Wooldridge [73] they allow stepping away from exact programmatic specifications to outlining models as interactions of autonomous processes. As agents by definition are autonomous entities able to perceive their environment and act on it, multi-agent systems (MAS) fit extremely well for the task of simulating social behaviour.

Throughout the years a lot of models has been developed utilizing simple reactive agents (for instance, such famous ones as segregation [16] or wolf-sheep predation [45][70][71]). This sort of simulations are essential for understanding complex problems and studying underlying mechanisms of emergent social behaviour. Thanks to modern developments of high-performance computing (HPC) these models can be brought to and researched at truly enormous scale, helping to answer important scientific questions. For instance, the Blue Brain project [2] (now part of the European Human Brain project [4]) aims at simulating brain at various scales as interactions of cells, molecules, etc.

There is a number of other popular applications of agent-based systems and models. One the classical ways of applying agent-based coordination and control models is in the field of robotics. Indeed, embodying agents seems quite a natural thing to do. Such implementations usually showcase different cooperation, coordination, and communication algorithms. For example, some of the robotics applications are described in [51][63].

Another hot topic for agent-based simulations is the rapidly developing area of private energy markets. In such markets it is possible to find both energy producers and consumers (and hence, referred as prosumers), which can lead to interesting behavioural patterns. To help study possible equilibriums of such

systems with huge numbers of participating prosumers, massive agent-based simulations seem to be one of the top choices [18][21].

Besides that there is a lot of areas where agent-based simulations and modeling could be used as one of possible tools. Such areas include, but are not limited to the analysis of supply chains [37], sensor networks design [69], agriculture[48], education [61], etc.

On the downside, most of these models use simplified reactive decision making methods where agents are simple rule-based or functional input-to-output transformers. A promising natural way to enhance agent-based simulations would be to imbue agents with further reasoning capabilities. An approach which aligns with current computing trends is to use (deep) reinforcement learning-based agents. With the current level of development of HPC such approach seems viable even for massive simulations (for example, OpenAI Neural MMO [62]). However, this way of simulating intelligence is far from the more fundamental reasoning approach: this sort of agents can learn from the data, but cannot reason without it, plus it makes it hard to explain the source of any decision made.

The classical way to model the reasoning capabilities in agents is to design them to be goal-oriented. Following the Belief-Desire-Intention (BDI) model [30] is the common way of doing this. In a BDI approach agents perceive their environment and extract some knowledge from it (represented in form of their **beliefs**). Then, based on it they can decide *what* they want to achieve (in terms of their **desires** and **intentions**) and *how* to achieve that (through some means-ends reasoning). These perceive-reason-actuate cyclic behaviours are supported by various planning techniques and reasoning models.

As mentioned before, this traditional approach to intelligent agent systems is better aligned with the sort of complex reasoning we expect to emerge from the interaction of sets of smart entities (for instance, GOAL [39], 2APL [25], Jason [11], etc.). However, what these systems lack is scalability. They are usually sequential by design and thus, taking into account the computational intensity of true logic-based reasoning, it is hard for them to scale efficiently to large numbers of agents.

In this work, we address the aforementioned scalability issue for BDI agents. Done in collaboration with Knowledge Engineering and Machine Learning Group (KEMLG) at UPC, the High Performance Artificial Intelligence research group at Barcelona Supercomputing Center (HPAI BSC), and, at the first stages, with the Smart Python Agent Development Environment (SPADE) team at Universitat Politècnica de València, we aimed at using BSC-based COMPSs HPC distributed computation framework in conjunction with the SPADE platform [33]. After the problem and systems' analysis, the final pro-

ject of the system was a custom Python-based BDI-agent simulation framework which respectively used COMPSs framework [44] to allow efficient and effective scalability on clusters.

The rest of this chapter is organized as follows: first, we state the given problem in § 1.1; after that we enumerate and describe the objectives of this work in § 1.2; then we outline the structure of this thesis in § 1.3, and finish with a brief summary in § 1.4.

1.1 Problem statement

We address the problem of creating a scalable BDI-based microsimulation platform. Microsimulations are a type of modelling complex social systems by simulating the individual actors within them.

As stated in the previous section, although there is a number of agent platforms supporting sophisticated reasoning capabilities in agents, they suffer from high time complexity. As the number of agents grow, the overall performance starts to suffer, with as little as a few dozens of agents, depending on the model.

Even though the most popular way to support massive agent simulations is to simply use reactive agents, often that is not enough to model complex individual behaviors. Therefore, with the goal of enabling large simulations of individuals with significant reasoning power, we consider the problem of adapting BDI-like agents to the domain of high-performance computing (HPC).

Developing MAS in the HPC domain seems rather natural, as these are intrinsically distributed. The challenge comes when one needs to support a coordinated and entangled simulation based on such system. Though various processes can be efficiently computed in parallel in the context of MAS, synchronization and keeping a balance among individual agent computations becomes an issue. The designer should also enable efficient ways of communication, while guaranteeing the existence of fail-safe methods for perceiving the environment without the risk of errors and deadlocks.

Thus, it is imperative for us to design the system that deals with all of these issues.

1.2 Objectives

The objectives of this work are:

1. Study the COMPSs framework for distributed computing and its use for implementing a microsimulations platforms under it.

2. Study the SPADE agent platform to assess its applicability in the HPC domain.

3. Develop a general multi-agent-based simulation model for the proposed framework.

4. Introduce a reasoning cycle to it.

5. Implement an agent platform based on this model.

6. Test the system in terms of performance and develop test simulations.

Since the project was done in collaboration with BSC, the main candidate for the HPC framework to support the system was COMPSs [44]. It is rather intuitive, supports parallelization of serial programs, and it was easily available for the testing purposes. Therefore, the first objective was to thoroughly analyse it and assess if it was possible to design a MAS platform with it.

Another collaborating party of this project was the SPADE team from Universitat Politècnica de València, therefore the main candidate for the agent platform for the framework was SPADE [33]. As COMPSs has a Python version (PyCOMPSs [65]) and SPADE is a Python-based package, both seem like good candidates for integration. Conveniently, SPADE provides almost all the functionality we need for the framework from communications to BDI module.

After the necessary framework studies, it was possible to move on to the main objective: creating the microsimulation model based on that framework. Here we needed to address the following design questions:

- Simulation cycle structure;
- Agent communication mechanism;
- Environment model and the perception and actuation mechanisms;
- Reasoning cycle in agents;
- Robust data transition mechanisms.

After coming up with the design choices resolving these issues, we needed to implement the model, translating the generalized model into executable code.

Finally, the system needed thorough testing. We had two testing objectives: the first one was to test its computational performance and the second one was to implement a complex microsimulation in this system.

1.3 Structure

The thesis document is divided into 6 chapters and an appendix.

Chapter 1 is this introduction. In this chapter we have outlined the purpose of this work, the problem we attempt to solve and the objectives we set to tackle.

Following that, in chapter 2 we present the analysis of various works related to our project. We present different agent languages and platforms and their approaches for MAS structure and simulation patterns. We also describe how these different languages and platforms implement reasoning in agents. Finally, we present some examples of HPC frameworks that can be used for the task, and some highly scalable agent platforms.

After that in chapter 3 we present our model. First of all, we describe the COMPSs framework and how we used it to support the simulation. After that we present the analysis of SPADE platform and explain why this platform was not used in the final system. Then we formalise the model and describe its main principles in terms of transitional rules. In addition to that, we describe the hierarchical task networks-based planner model that we used as a main reasoning engine for the system.

Chapter 4 describes technical aspects of the implementation of the proposed model. It presents the structure of the Python project, the user API, and instructions for designing custom simulations using the system.

After describing the model and its implementation, we detail the experiments we have conducted in chapter 5. We outline the plan of these experiments, their purpose and expected results, and then show the experimental results achieved after testing the system on BSC clusters.

Finally, we sum up the results we have achieved in the conclusions in chapter 6. We comment on each objective we had, and discuss the future work and enhancements that are in progress.

Important parts of the system's code are also provided in the appendix A.

1.4 Chapter summary

In this chapter we have introduced the agent-based approach for modelling complex systems and explained why we need to combine it with HPC approach. We have also stated the main objectives in our task of developing high-performance agent-based framework and pinpointed the main candidates for the foundation of the system: COMPSs framework and SPADE platform.

In the next chapter we will present related research work, show and describe papers essential to our work, as well as various similar past and ongoing research projects.

Chapter 2

Related work

In this chapter we discuss the state-of-the-art in the relevant areas is discussed. A number of relevant papers are also discussed, such as some fundamental works in MAS and their possible applications.

The chapter is organized as follows: first, we cover the area of multi-agent systems and simulation platforms in § 2.1; secondly, we go over some high performance computing technologies relevant to this work in § 2.2; thirdly, in § 2.3 we cover the papers about agent planning.

2.1 Multi-agent systems and simulation

One of the most fundamental works on MAS is Michael Wooldridge's "An Introduction to MultiAgent Systems" [73]. This book focuses on the concept of an agent, its possible properties, types, interactions, etc. Additional aspects of agents and MAS are also covered in another fundamental work: "Artificial Intelligence: A Modern Approach" by Stuart Russell and Peter Norvig [57], and in a number of various surveys and research papers, for instance [9].

By definition of these sources, an agent is a computer system capable of independent action on behalf of its user or owner (figuring out what needs to be done to satisfy design objectives, rather than constantly being told). Multi-agent system is one consisting of *interacting* agents.

The aforementioned sources for agent definitions name a number of various agent properties. Some of them have proven to be essential over time (for instance, autonomy), some were found to be inapplicable in the real-world scenarios (for instance, mobility). As per these definitions, the key properties of agents for our system are:

Autonomy: Agents need to act on their own without users directing

their every step.

Flexibility: Agents need to be reactive, proactive, and social (see below).

Reactivity: Agents need to respond to changes in their environment.

Proactiveness: Agents need to attempt to achieve their goals.

Social ability: Agents need to interact with each other.

Rationality: Agents' actions should move them towards achieving their goals.

Reasoning capabilities: Agents should be able to reason about their environment, their goals, and actions, and plan possible courses of actions.

Besides these desired agent properties, we also need to mention the agents' internal architectures. There are 3 main architectures proposed in the bibliography: reactive, deliberative, and hybrid. As it was mentioned in the chapter 1, most of massive agent-based simulations use reactive architecture for agents, due to its simplicity: reactive agents just react to changes in their environment in a functional manner. They have only the autonomy, reactivity, and social ability properties (JADE [10] is an example of agent platform featuring simple reactive agents). On the other hand, deliberative agents tend to model the environment and do complex planning rather than simply acting on every change (for instance, GOAL [39] agents are deliberative ones). Hybrid architecture, however, is a middle-ground: hybrid agents behave reactively to some changes in the environment, while keeping a reasoning cycle that can be interrupted to reacto to meaningful input (for instance, SPADE [33] or JadeX [13] agents may be called hybrid ones when powered by BDI engine).

In the BDI model[30][54][1], the agents are goal-driven and are defined by 3 main concepts: a set of **beliefs**, representing their knowledge of the world, a set of **desires**, representing what they want the world to be, and a set of **intentions**, representing what agent plans to do in the immediate future. Depending on the architecture, the agent perceives the environment periodically, updates all or some of these sets, and decides how to act. This is called BDI reasoning cycle (see Fig.2.4 for an example of an advanced cycle).

Another extremely important part in formalising agents is played the Foundation for Intelligent Physical Agents (FIPA) [3]. For more than 20 years of

[1] it was already mentioned at the beginning of Chapter 1

its existence FIPA has introduced an extensive set of standards for agents, including architectures, protocols, and communication standards. The latter, in form of their Agent Communication Language (ACL), is arguably its main contribution to the MAS community, as it allows agents of different origins, architectures, and underlying code to interact with each other and has been widely adopted by the MAS community. For instance, FIPA ACL defines the following structure of the messages:

performative: the type and the purpose of the message

sender: id of the agent who sends the message

receiver: id of the recipient of the message

content: the actual content of the message

reply-to: id of the agent to whom to send a reply

language: language of the content of the message

encoding: encoding of the content of the message

ontology: ontology used for the content of the message

protocol: the protocol that this communicative act follows

conversation-id: the unique identifier of the communicative act this message is a part of

reply-with: identifier of the response that should be used for this message

in-reply-to: the reply-with identifier of the message this message is replying to

reply-by: deadline for the reply

As it can be seen, the structure is quite extensive and aims to cover all the possible situations in open environments. It is an open question whether all of these fields are needed for simulations, as simulated scenarios are usually closed environments (where there are no third-party agents that can be added at runtime).

One of these fields in ACL that is crucial in almost all multi-agent systems is **performative**. FIPA specifies a list of standard ACL performatives that

cover a broad range of possible agent's intentions, such as *inform*, *propose*, *agree*, *query-if*, *request*, and many others. These performatives make it easier to model the communicative acts and simplify the message structure by explicitly stating how the message's recipient should interpret its content. For example, if message's content is "open door" (given that these words are defined in the used ontology), its meaning will be defined by the performative: request to open the door in case of "*request*", question about the door's state in case of "*query-if*", or notification about the door's state in case of "*inform*".

2.1.1 Agent languages and platforms

There is a significant number of agent languages. Some of them are general purpose, some are designed with a specific problem in mind. Moreover, there is even more different multi-agent platforms and software systems designed for the simulation purposes. In this subsection we will cover some of the most important of them, as well as a number of surveys that provide additional information on them. A more extensive list of Agent Languages and platforms can be found, for instance, in [42].

JADE [10] is an agent platform that was designed with FIPA standards in mind with the purpose of creating a platform to act as a middleware for agents of different origins. The platform is written in Java and provides 3 important components: Agent Management System (that handles the agents' lifecycle as a part of MAS), Directory Facilitator (that provides agents with information about other agents in the MAS), and Agent Communication Channel (that allows agents to send each other messages). The platform is designed to work in a distributed environment connecting different instances of itself and facilitating agent distribution among these instances. The model behind JADE is shown on Fig.2.1.

Though JADE is presented as an agent platform, it does not specify the internals of the agents that inhabit it. Still, JADE agents are reduced to user-defined functions on how to react to different stimuli, thus, making most of their implementations limited to reactive behaviours without adoption of external tools.

2.1.1.1 BDI-based and BDI-supporting agent languages and platforms

One of the advanced languages based on the JADE platform is JadeX [13]. It extends JADE to introduce a number of useful concepts, such as functional (web)services, available across the specified network. The most important

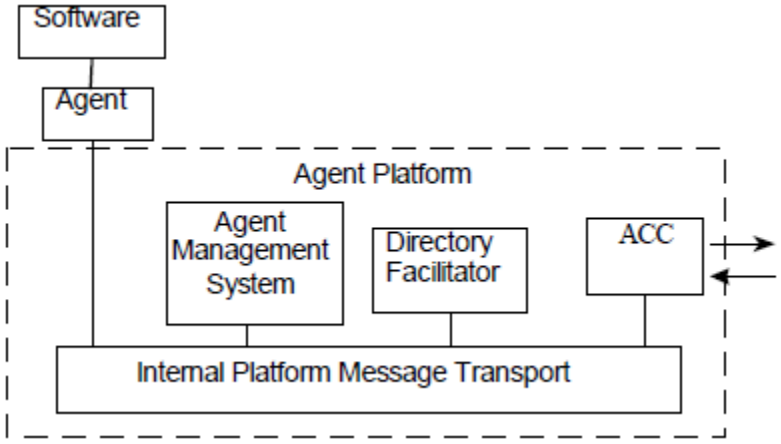

Figure 2.1: JADE platform model

JadeX's addition to JADE agents is a BDI module. Though not utilising first-order logic as some of the languages and platforms mentioned below, it provides the concepts of Beliefs, Goals, and Plans, and basic reasoning capabilities to select goals to follow and plans to achieve them. The structure of the reasoning cycle of JadeX's BDI agents is shown on Fig.2.2.

SPADE [33] is an example of agent platform that uses an external framework to support its functions. This platform is heavily based on Extensible Messaging and Presence Protocol (XMPP) (and messaging systems that utilize it) to support a variety of functions. As XMPP provides such services as presence manager, messaging, and support of various communication lifecycle events, it is extremely well-fitted to be a base for agent platform. In fact, because of this SPADE agents have the ability to connect to normal chats that follow XMPP protocol, and thus interact with real people.

Moreover, previous version of SPADE provided first order logic-based BDI module to support reasoning in its agents (BDI module for the current version of SPADE is still in development).

Another important agent language for this project is `CANPlan`. One important feature of this high-level agent language is the integration of BDI model with hierarchical task networks (HTN) planning. HTN planning [31][32] is a rather straightforward and lightweight planning model which uses hierarchies of abstract and primitive tasks to represent some domain knowledge of different behaviours and methods of acting. `CANPlan` shows that BDI and HTN planning are compatible with latter being a valid way for BDI agents to carry out look ahead planning.

2APL [25] language is designed around the BDI model and provides an

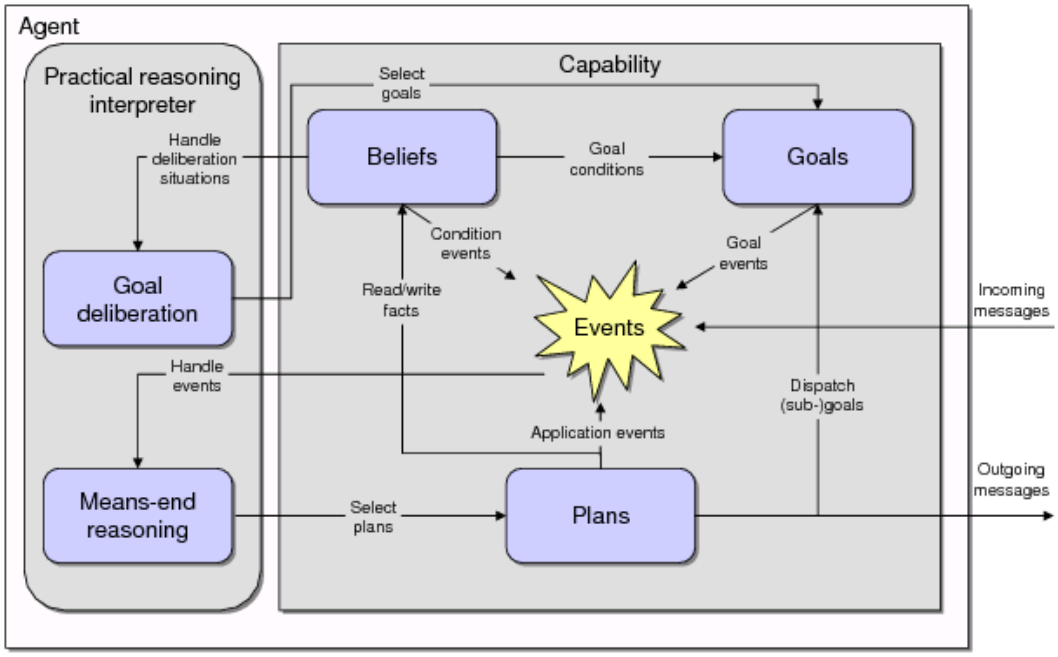

Figure 2.2: JadeX reasoning model

extensive agent description syntax and formalism, as well as the usual concepts of plans, goals and beliefs. It also specifies various rules that allow agents to reason about various events, analyze and execute plans, etc. The deliberation cycle for 2APL is provided on Fig.2.3.

Jason [11] is an implementation of AgentSpeak language, also designed with BDI model in mind. The main idea behind it was to mix declarative and imperative programming approaches to allow for normal logic-based reasoning while still giving user enough control over the ways agents act within different plans. Its internal reasoning engine processes agents' beliefs, goals, and available actions into executable plans. Jason's BDI reasoning cycle (Fig.2.4) is a perfect example of such, covering different important aspects of generation of plans and replanning.

Goal [39] is an agent programming framework that uses classical STRIPS-style planning [34] in agent BDI model. Namely, it defines the notions of domain knowledge, beliefs, and goals and use them as, respectively, pre- and post-conditions for STRIPS planner.

Another notable example of agent language and platform is JACK [72]. Similarly to JadeX, JACK uses event-based execution cycle, using beliefsets to choose among a set of user defined plans (that are either plain code, or

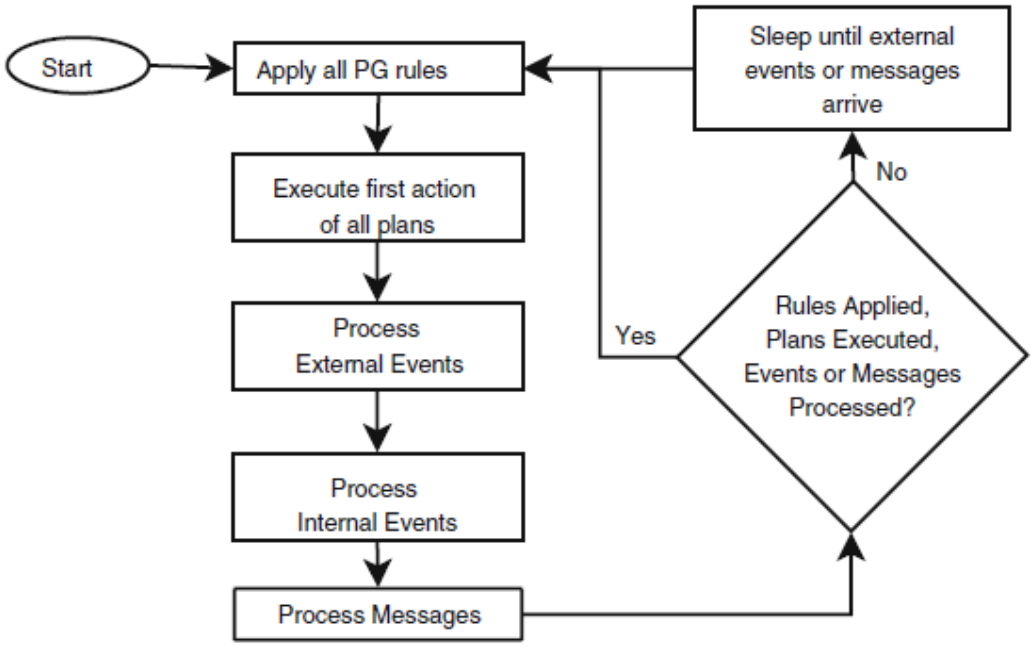

Figure 2.3: 2APL individual agent deliberation cycle

commands in JACK language). The most important contribution of JACK however is its suite of methodological tools. These tools allow to fully embrace the agent approach as a next programming abstraction level, designing programs as systems of interacting agents, with their goals and beliefs, instead of concrete user-defined interactions of objects. Using JACK's tools it is possible to define the high-level agent-based system definition, and let JACK to generate lower-level code from it.

KGP [41][12][58] is BDI-based reasoning model and its respective implementation. Similarly to GOAL it uses concepts of knowledge base, goals, and plans to power the agents' reasoning capabilities. It uses event calculus to define an event-based reasoning, which allows agents to react to various processes of the environment in an advanced deliberative manner. The conceptual organization of its agent is presented at Fig.2.5.

Other examples of agent languages and platforms with BDI capabilities include: MINERVA [43], METATEM [35][36], Golog family [59][27][28], etc.

As we can see, there is a significant number of BDI-based (or at least, BDI-supporting) agent platforms. It is important to note that most of them use variations of the full BDI model, with most changes concerning the concepts of desires and intentions. Another issue from the point of view of our

project is that none of them support direct integration with supercomputer clusters. Some, like JADE platform, though supporting distribution, are more suitable for systems of full-fledged computers as nodes, not a number of computation nodes of a cluster. This makes them an unlikely choice for the HPC applications.

2.1.1.2 Scalable and HPC-based agent languages and platforms

Unlike the agent languages and platforms that support heavy-weight reasoning mentioned in the previous section, this section focuses on platforms for high-scale agent computations. Such platforms use HPC features and efficient MAS models at cost of high-end reasoning capabilities.

MASON [46] platform and its parallel version D-MASON [24] implement grid-based and continuous space simulation environment for agents. In order to ensure parallelism D-MASON uses master-worker approach to distribute portions of simulation space to a number of workers (Fig.2.6). For synchronization, workers exchange possible agent interactions with their in-simulation neighbours. For communication, they use publish–subscribe design pattern, with agents publishing information and messages into their regions' channels and other agents subscribing to these channels. Moreover, to support reproducibility, agents update their states based on the states of their neighbours in the previous step (thus, eliminating interdependencies). Finally, to deal with heterogeneity of workers, D-MASON can create multiple "virtual" workers on more efficient physical workers.

FLAME [20] platform offers another approach to parallelization of MAS. It uses Communicating Stream X-Machines Model, where each agent is represented as a state machine of a specific structure. Each state has a function associated with it and some of these state functions can, once per iteration, read or write messages of a certain type from the corresponding message board (Fig.2.7). Each agent execution in between reads and writes can thus be parallelized and only the message boards need to be synchronized. To increase the efficiency of the computations even more, the special FLAME scheduler analyzes the state function interactions and schedules their execution in such way that maximizes the interval between write and read functions.

One of the most scalable and efficient platforms for agent-based simulations is RepastHPC [23][22]. It uses scheduler for each memory-independent process and gives user very precise control over the scheduling of agents, their functions, and data synchronizations. However, this is rather demanding in terms of the required skill of users. In fact, there is a lot of precomposed scheduling models for various existing simulations (for instance GridABM [38]

presents some communication schemas for RepastHPC), though for each new simulation a new model must be created.

Finally, there is another scalable agent platform designed to handle parallel agent execution, PANDORA [56], developed by BSC. Pandora, as most of the other such platforms, utilises spatial model of the environment. With this assumption, the natural way of distribution of computations is to divide the environment into several sections, each handled by a separate computation node. The in-node computations are organized in such way, that agents can run in non-intersecting groups, and between-node interactions though costly are acceptable.

It is important to note that PANDORA has a very limited support for reasoning capabilities in agents which excluded it from a list of potential candidates for agent platforms we could have used for our system.

There are other platforms that support reactive agent simulations, though they are not designed to work in the HPC domain. Even though, depending on the implementation and model used, such platforms may perform well even on normal desktop computers, they will eventually get outperformed by those designed for HPCs. An extensive study of various agent simulation platforms has been done in [7]. This paper analyses 85 agent simulation platforms and toolboxes including those mentioned above, presenting their characteristics, such as language, scalability, difficulty of the language, domain, etc. However, most of them are not designed with HPC in mind and thus are not intrinsically scalable (for instance SWARM [49], PDES-MAS [67], and NetLogo [5][68]). Due to this limitation such platforms are not included in the scope of this chapter.

Another nice study on agent simulation platforms is presented in [53]. Although it is not quite as extensive as [7] (but still thorough for its time), it provides valuable feedback and some guidelines for the development of custom agent simulation platforms, such as addressing the development tools complexity, IDEs integration, need for basic statistical tools, etc.

There is a number of other surveys done and available. As their intersection rates are high, we will simply mention some of them with their main contributions. In [55] authors study different HPC agent platforms and compare them using a set of devised requirements. [8] is an older survey covering and analyzing different aspects of 55 agent platforms, packages, and systems. [42] studies a total of 24 platforms, comparing them by such criteria as operability, usability, pragmatics, and security management. [50] studied a total of 54 platforms and provided various information on them, such as domain, license, language, etc.

```
1.  B ← B0;        /* B0 are initial beliefs */
2.  I ← I0;        /* I0 are initial intentions */
3.  while true do
4.    get next percept ρ via sensors;
5.    B ← brf(B,ρ);
6.    D ← options(B,I);
7.    I ← filter(B,D,I);
8.    π ← plan(B,I,Ac); /* Ac is the set of actions */
9.    while not (empty(π) or succeeded(I,B) or impossible(I,B)) do
10.     α ← first element of π;
11.     execute(α);
12.     π ← tail of π;
13.     observe environment to get next percept ρ;
14.     B ← brf(B,ρ);
15.     if reconsider(I,B) then
16.       D ← options(B,I);
17.       I ← filter(B,D,I);
18.     end-if
19.     if not sound(π,I,B) then
20.       π ← plan(B,I,Ac)
21.     end-if
22.   end-while
23. end-while
```

Figure 2.4: Jason BDI reasoning cycle

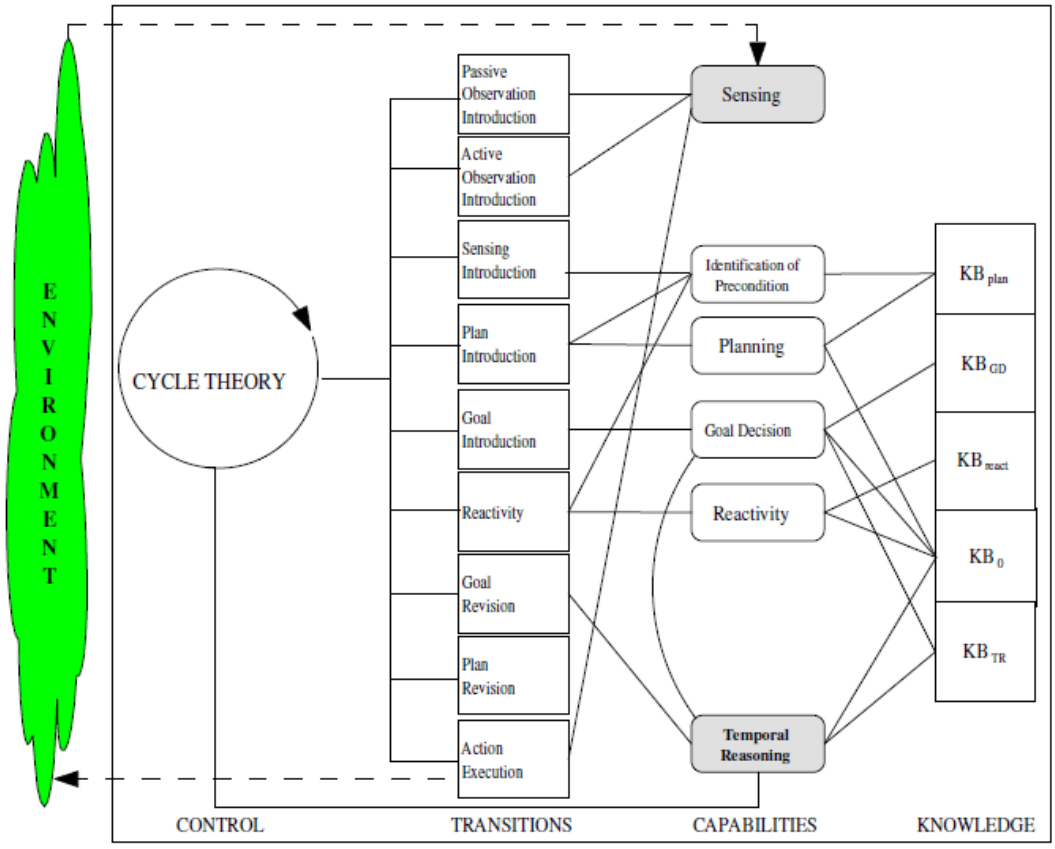

Figure 2.5: Conceptual organization of KGP agent

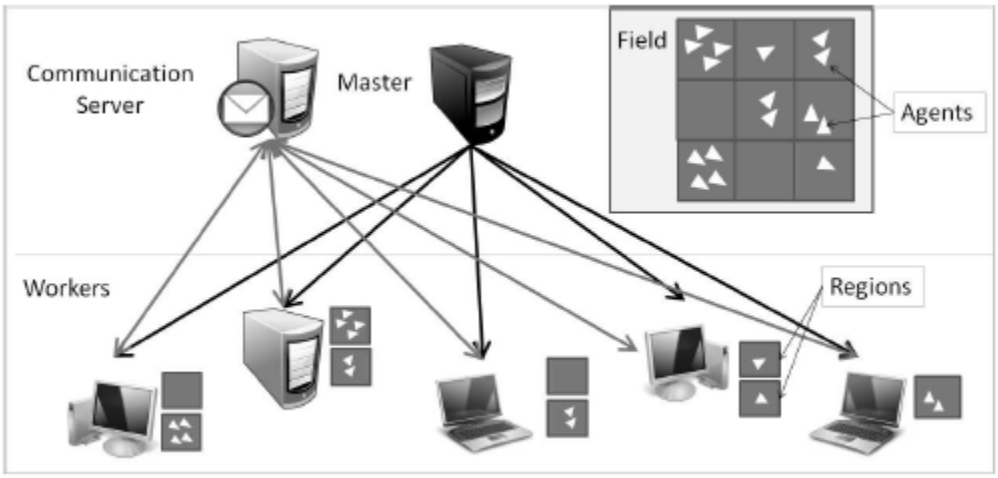

Figure 2.6: MASON functional blocks

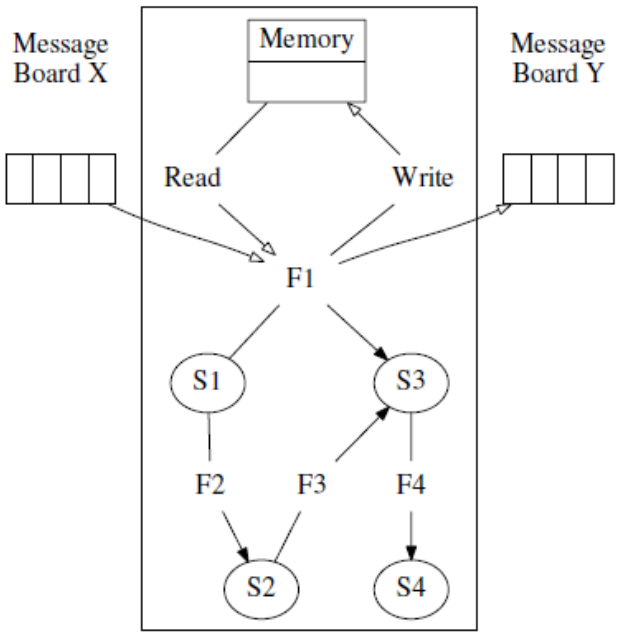

Figure 2.7: FLAME agent example description

2.2 High-Performance Computing

As HPC-based agent platforms, such as MASON [24], FLAME [20], PANDORA [56], and RepastHPC [23], were covered in the previous section, in this one we focus on the HPC technologies that allow massive agent simulation.

The central HPC framework for this project is COMPSs [44] and its corresponding Python package, PyCOMPSs [65]. Though details of COMPSs execution cycle are provided further in chapter 3, we will note some of its key features. COMPSs HPC framework is quite different from HPC systems shown in § 2.1: instead of providing tools to develop explicit distributed model, it uses a series of code instructions to transform a normal serial program into a distributed one. This additional layer of abstraction allows its users to focus on designing the program workflow in terms of parallel processes instead of dealing with machine-specific distribution mechanisms[2].

Though COMPSs framework works great for the purpose of supporting HPC computations, another important issue we need to address is robust and efficient data transfer. With calculations being distributed over a number of nodes (and sometimes carried out by different nodes at different times) we may need to have some data (like shared environment) available at different parts of cluster. dataClay [47] is a tool designed specifically for this task for various distributed systems, including clusters. It uses rather simple programming interface and with a few additions to the code it is possible to ensure persistence of various objects across the whole system.

Another interesting aspect of HPC-based agent simulations is studied in [74]. The authors address the issue of how different HPC hardware and technologies can help different aspects of massive agent-based simulations. Specifically, they cover many-core CPUs, GPUs, Accelerated Processing Units (APUs), Field-Programmable Gate Arrays (FPGAs), and other specific hardware types. As important challenges of agent-based simulations in HPC they note the following ones:

1. Hardware assignment
2. Data transfer overheads
3. Scattered memory access
4. Maximization of parallelism
5. Abstraction from hardware specifics

[2] An extensive analysis is presented in § 3.1.

For each of these challenges authors provide literature analysis of possible solutions. Thus, this paper can be considered quite valuable for designing low-level aspects of scalable agent simulation platform.

2.3 Planning

Most of BDI-based agent languages and platforms include plans in one way or another within their reasoning cycle. Some may use plans as user defined functions, containing some plain code (JadeX [13], Jack [72]), other use logic-based formalisms to define them as a series of actions (as preconditions to postconditions transitions) (Goal [39], KGP [41], IndiGolog [28] of GOLOG family).

The latter is one of the most fundamental planning approaches, with the STRIPS planner [34] being of the oldest planners out there. The basic idea is to define our environment (and its possible states) in terms of predicates, and to define a set of actions; each of these action must have a set of preconditions (in terms of predicates that can be matched against the current state) and postconditions (the changes to the environment that become true after that action is performed). With that information STRIPS planner tries different combinations of actions until it is able to finally get to the goal state (or until some stop condition is satisfied, for instance, the time is out, or the maximal search depth is exceeded).

Hierarchical Task Network (HTN) planning [31][32] is another approach to the task. The main idea is to have a hierarchical decomposition of various tasks in our domain, starting from the most general, abstract ones and finishing with primitive tasks. While decomposing compound tasks we may use some conditions for tasks, as well as different quantifiers for their children (either one of them is needed or all of them). In the end we get a structure, by decomposing which we will get a series of primitive tasks, or actions, to perform: a plan. This approach requires the use of domain knowledge to design HTNs, however, the plan generation in this case becomes rather trivial. This simplicity made HTN planning quite popular choice for a number of agent languages. CANPlan [60], for instance, as mentioned earlier, is a high-level agent language that incorporates HTN planners in the agents' BDI cycle. Another example is [40] — a computer game interpretation of HTN planning, adapted for easy online plan generation in real-time agent environment.

Some other interesting planning techniques are worth mentioning. [17], for example, provides a model for multi-agent social planning used to achieve collective social goals (and is also using HTNs for the generation of these

plans). [14] introduces a concept of Continual Planning when agents' planning cycle gets deliberatively postponed to some later stages of simulation when the updated information about the environment is gathered.

2.4 Chapter summary

In this chapter we have discussed various relevant research papers. We have started with some of the most fundamental works in MAS field and introduced important concepts of agents and multi-agent systems. Then we moved on to overview of various agent languages and platforms describing their features, focusing on the most important ones for us: BDI reasoning support and scalability. Finally, we have covered some planning approaches in agent platforms, such as classical STRIPS planner and HTN-utilizing planners. The latter seems like a good candidate for a reasoner implementation for scalable simulation platforms.

We have also found out that as of now BDI reasoning and scalability are mutually exclusive features in agent platforms: it is possible to either have at most a hundred of complex BDI agents in some platforms, or thousands of reactive ones in the others. Moreover, easy cluster deployment is a feature of only some of the scalable agent platforms. And for a set of platforms with BDI agents this is impossible by design (notable example of them is SPADE).

In the next chapter we will present a deep analysis of COMPSs and SPADE as candidate frameworks for HPC and agent reasoning.

Chapter 3

Proposed model for BDI agent-based HPC microsimulation framework

In the previous chapter we have discussed relevant works. First, we have covered various agent platforms and, specifically, ones that use BDI and BDI-like reasoning. We have also described different agent platforms and general HPC approaches for agents. Finally, we have seen planning from MAS perspective. In this chapter we introduce our own agent model.

The purpose of this model is to allow the efficient use of reasoning agents on HPC systems. This will make it possible to run large-scale microsimulations of complex domains (such as, for instance, large-scale traffic simulations).

The chapter is organized as follows: in § 3.1 we introduce the COMPSs HPC framework and the corresponding Python package; § 3.2 briefly covers the analysis of SPADE platforms and justifies why this platform has not been used in the final system; then, in § 3.3 we explain the design of the proposed model and the workflow approach we use; finally, in § 3.4 we explain the used reasoning model and tools we provide in addition to the system.

3.1 COMPSs framework

COMPSs [44][64] is a framework based on the Grid Component Model [6] and ServiceSs model [66] developed by BSC. Its main purpose is to allow the development of distributed cloud- or grid-based applications without the need of dealing with the specifics of underlying systems that will execute them. Thus, it provides a layer of abstraction, allowing the creation of hardware-

unaware applications (which will be later automatically distributed), saving the developers from the effort needed to understand the low-level details about the used hardware. In other words, it improves programmability.

There were several rationales behind prioritising the selection of COMPSs as the main candidate for the HPC part of the designed agent simulation framework. The first one, as it was mentioned in the beginning of Chapter 1, is that this project is a collaboration between UPC and BSC; in this context, COMPSs framework is easily available, as well as its support.

The second one is the versatility of COMPSs: it is designed to be used on any sort of clusters and cloud platforms (which already gives it a lot of possible development and deployment options) and can also be deployed on desktop computers via the use containers (these can actually be united into a virtual cluster).

3.1.1 COMPSs runtime

From the point of view of the user everything is straightforward: you need to write a normal sequential program as usual and then add some annotations (i.e., decorators) to functions that might be run in a distributed manner. At runtime COMPSs will analyze the data dependencies between these functions (called *tasks* as in the GCM model) and will run in parallel those tasks that are safe to be concurrently executed. Besides the basic functionality users also have control over how these tasks are handled by using constraints. In this way one may request only a certain type of nodes for some tasks, for instance, having a specific software library installed, having certain memory volume or processors number, or even requesting a specific processor. Also, users may force the distribution of tasks to different computation nodes in a round robin manner, or request that the task is replicated on all the nodes. Finally, in COMPSs one can also use cloud services that may be invoked and run as needed.

The core of this runtime is written in Java, but C/C++ and Python interfaces are also available. As all of these commands are language-specific, as well as the annotation style (while still being similar), we will provide some syntax examples for Python in Subsection 3.1.2.

COMPSs runtime architecture, as per GCM model, is divided in several components (Fig.3.1): Task Analyzer, Data Info Provider, Task Scheduler, Scheduler Optimizer, Resource Manager, Job Manager, and File Transfer Manager.

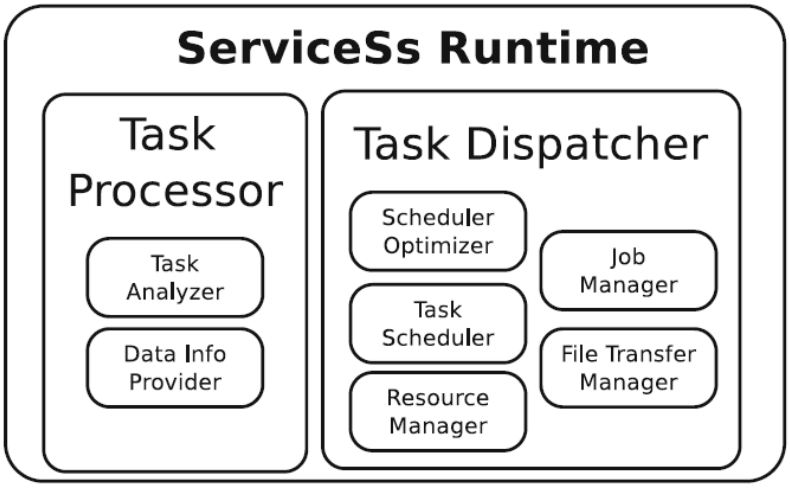

Figure 3.1: COMPSs runtime structure

Task Processor is composed by *Task Analyzer* (TA) and *Data Info Provider* (DIP). These are the components that directly interact with user code. TA goes through the tasks, analyses user-specified flow of data between them, constructs an acyclic dependency graph (see Fig.3.2 for an example), and uses it to send tasks to the scheduler. The data dependencies information is gathered with the help of DIP: it is responsible for creating separate copies of data each time the corresponding variable or object is written to in code and tracking which piece of code needs which version of it.

The dependencies graph built by TA is updated in an online fashion and when TA detects that some task got free of its dependencies, it sends this task to the Task Dispatcher.

Task Dispatcher is responsible for running the tasks. It consists of *Task Scheduler* (TS), *Scheduler Optimizer* (SO), *Resource Manger* (RM), *Job Manager* (JM), and *File Transfer Manager* (FTM). RM gathers and keeps up-to-date the information of all the computational resources that are available. Scheduler then uses this information to define how tasks should be distributed over them. It uses a scoring algorithm to define which resource is better for the task, mostly based on what data is needed for that task and what data is already available there. SO runs in parallel and uses more sophisticated scoring techniques, detecting possible improvements to the current schedule and modifying it as needed. Also, if some data on one resource node is needed at the other, FTM handles the transference.

When everything is ready to run the scheduled task, it goes to the JM.

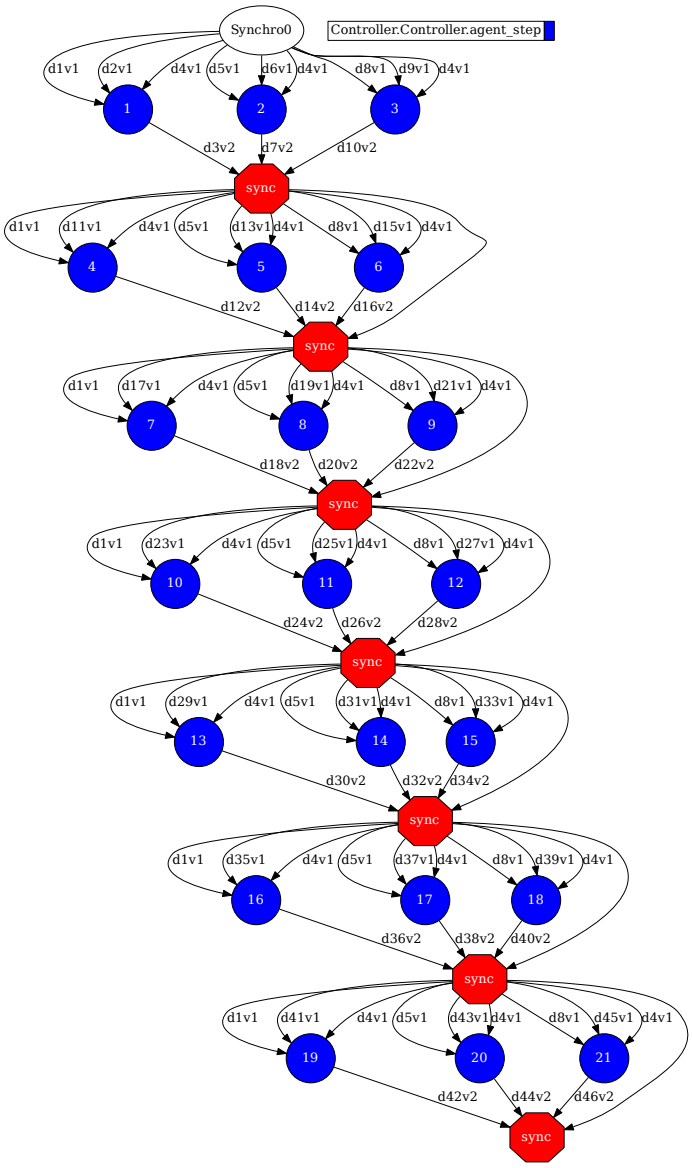

Figure 3.2: Example of a COMPSs dependency graph. Blue nodes are tasks, red ones are synchronization calls, arrows are data dependencies labeled with version numbers that DIP provides

After that it launches the actual computation job with the task code, provides the data, handles the code execution, and gathers and processes the results.

All these components run in parallel to ensure smooth user experience.

Note that due to possibly intense data transfers (though they are optimized a lot), COMPSs framework is not very well suited for fine-grained tasks, for them one can use the similar framework designed for the task: OmpSs [29][15]. COMPSs is feasible when the program's tasks are coarse-grain, requiring at least tens or hundreds of milliseconds while OmpSs tasks need to be from about microseconds to tens of milliseconds.

In the next subsection, Python-specific details are provided for better understanding of how the framework works.

3.1.2 PyCOMPSs

PyCOMPSs [65] is a version of COMPSs framework designed to deal with programs written in Python.

From the runtime point of view, it adds a few layers to the basic COMPSs Java runtime, as shown in Fig.3.3. Essentially, what these additional layers do is transform the Python data types into Java ones, for proper data dependency analysis by the main runtime. Python bindings gather these types of information by analyzing the code and using the auxiliary C++ library for proper binding and transformation.

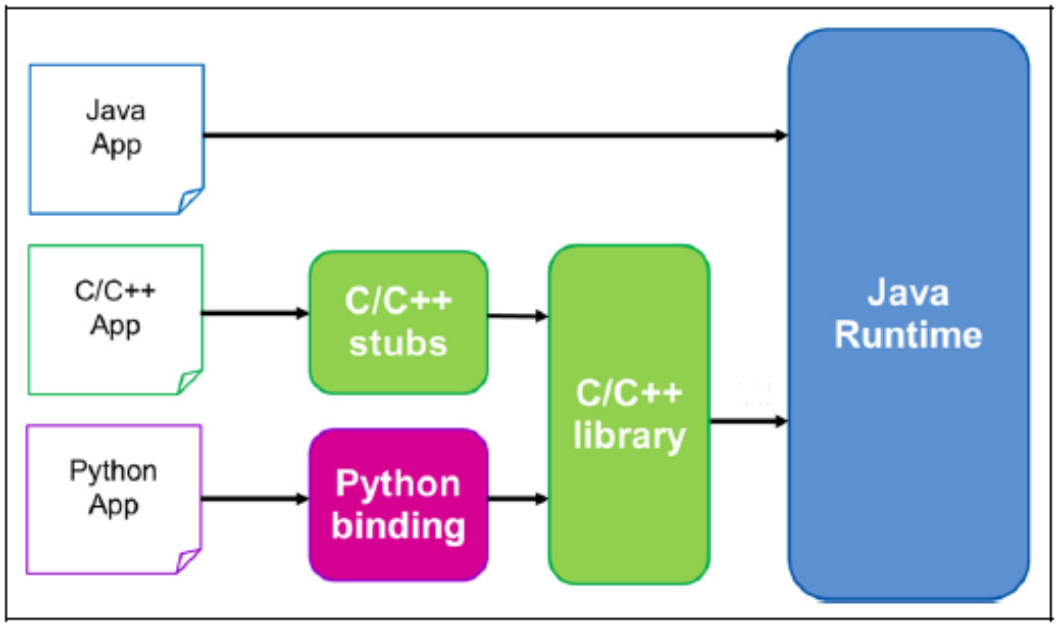

Figure 3.3: PyCOMPSs runtime structure

The harder part comes when a custom class object is used as a data type

in the program. The solution here is to avoid direct type casting by using COMPSs' support of files as possible inputs. This is illustrated in Fig.3.4. As it is shown, PyCOMPSs serialises the custom objects and passes them as files. When the time comes to execute the corresponding tasks, these objects are deserialised and passed to the code (which is regular Python code).

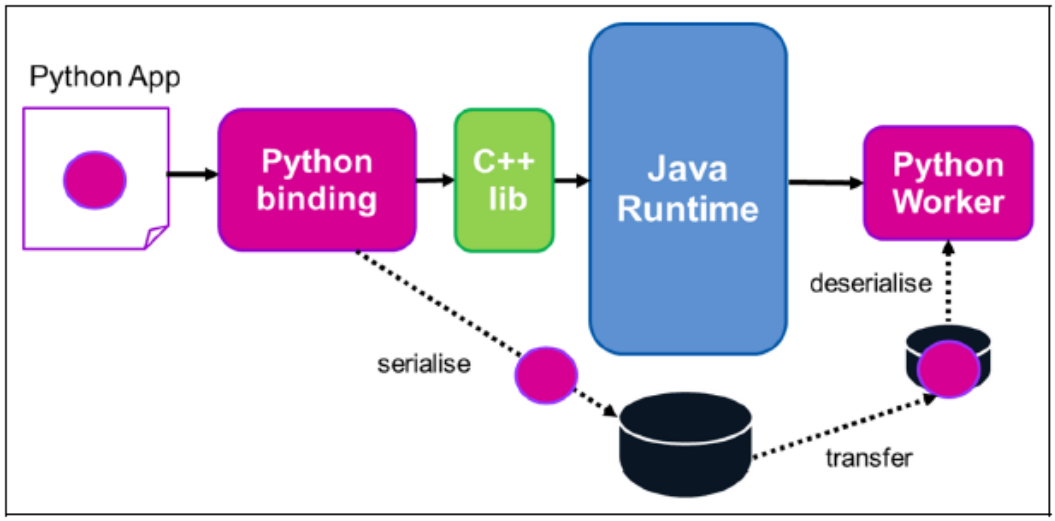

Figure 3.4: PyCOMPSs object handling scheme

Note, however, that this approach adds an additional requirement: the custom class objects must be serialisable. This will be covered in more detail in § 4.1.4.

Codewise, the changes to the original sequential code are minimal, as stated earlier. COMPSs instructions are done in the form of annotations and several API functions and procedures. The main annotation is `@Task(...)`. Here user must specify the return type (in form of type or class name) and the data direction of inputs for mutable types and classes (all immutable types are naturally read-only, the direction is IN, and by default all the inputs are also IN, so there is no need in specifying that).

Below is an example of a PyCOMPSs task annotation:

```
@Task(par1=INOUT, par3=OUT, returns=list)
def my_func(par1, par2, par3):
```

In this example we have a user function `my_func()` with 3 parameters:

par1 is specified as OUT which means that in `my_func` its value gets

overwritten without getting read (i.e. it is not necessary to pass it to the current node).

par2 is not specified, which means that this parameter is IN; its value is used, but is not modified (i.e. COMPSs do need to get its latest version to the node, but will not create a dependency for the following calls to its value)

par3 is specified as INOUT which means that its value is both read and modified.

And we also specify that the return type is `list` so that the JM can properly handle the output.

In this way we may mark global functions, class methods, and instance methods as tasks.

Besides the annotations, the two most important API functions that are provided by PyCOMPSs are `compss_barrier()` and `compss_wait_on(var)`. They are needed in situations when we need to have a level of control over the task execution and data synchronization.

- `compss_barrier()` function is used to block the execution of the containing Python script until all the submitted jobs finish. It is useful for instance to measure the performance. However, the resulting values for different variables are not synchronized with the main master node. In fact, when the PyCOMPSs Task Processor goes through the tasks, it replaces the actual outputs of the functions with either Future objects (for non-custom data types), or by default-valued objects of user classes.
- In order to get these values synchronised `compss_wait_on(var)` function must be used. We must specify what variable we want to synchronise and to which variable we want to write these values. This will also halt the execution until the latest version of the required data is calculated, which is then passed to the master node.
- In case we need file synchronisation instead of variable synchronisation, `compss_open` should be used instead of `compss_wait_on(var)`. It works the same way as the standard Python `open` function, and is used to wait for the latest modification of the requested file before trying to read or write to it.

Besides these functions and annotations, there is a lot of others, allowing to control various aspects of tasks assignment and execution, but they are

not used in the proposed system and, thus, are omitted from this description (more information may be found in the manual at [1]).

3.2 SPADE platform

As specified in § 2.1, SPADE is a Python-based agent platform based on the XMPP protocol. The package provides an extensive set of communication-based functions, as well as proper asynchronous agent processing. This platform is a really good example of open multi-agent system supporting distribution and possibility of parallel execution of agents. As SPADE agents "exist" on normal XMPP servers they can without any issues interact with actual people as well as with other agents (though it will require more strict messaging protocol enactment by participating persons).

XMPP protocol handling and asynchronous server-client interaction are managed by Python `aioxmpp` package. Parallel agent execution is provided by the popular `asyncio` package. The use of these packages is entwined in ost of SPADE's functionalities: messaging, directory facilitation (handling the list of agents, their unique identifiers, and providing this information on demand to other agents), and presence management.

Despite the great range of possibilities this setting provides for applying SPADE in real-world, this configuration also sets a critical problem for HPC application: XMPP protocol is designed as a messaging protocol for a number of full-fledged systems and is not available as a mean of communication between computation nodes or CPUs on clusters. `asyncio` is also not an option for a framework if COMPSs is used, as the latter provides its own means of asynchronous execution. Moreover, the BDI functionality is still under development in the updated SPADE version (and the previous version is not compatible with the current one and depends on explicit threading control).

These complications would have forced us to reimplement most of the SPADE functionalities which would have proved to be suboptimal at the very least. Because of this we have decided not to use SPADE in our system. Instead, we have proceeded to use simple custom agent system which was then developed to its current version.

3.3 Multi-agent system

In this section we will introduce the formal conceptual model of our agent-based HPC simulation framework. Its main purpose is to present the main elements of the framework and formally describe their operational semantics.

Let us define our multi-agent system $\mathcal{M}$ as the following tuple:

$$\mathcal{M} = \left\{E, \mathcal{A}^+, \mathcal{C}\right\} \tag{3.1}$$

where:

- E is an environment, a conceptual entity where the agents are "located", which they can perceive, and on which they can act
- $\mathcal{A}^+$ is a non-empty set of agents
- $\mathcal{C}$ is a controller, a special task responsible for maintaining the environment and facilitating communications between agents

An agent $\mathcal{A}_i$ can be defined as the following tuple:

$$\mathcal{A}_i = \{ID, msgQs, Bh, \mathbb{B}, \mathbb{G}, \mathcal{P}, outAcs\} \tag{3.2}$$

where:

- $ID = \{AgID, AgDesc\}$ is $\mathcal{A}_i$'s identity data
  - $AgID$ is a unique identifier of $\mathcal{A}_i$
  - $AgDesc$ is an arbitrary description of $\mathcal{A}_i$
- $msgQs = \{\mathcal{I}, \mathcal{O}\}$ is the queue $\mathcal{A}_i$'s messages
  - $\mathcal{I} = \{\dots, msg_i, \dots\}$ is the Inbox, a set of messages sent to $\mathcal{A}_i$
  - $\mathcal{O} = \{\dots, msg_i, \dots\}$ is the Outbox, a set of messages sent by $\mathcal{A}_i$
  - $msg_i = \{AgID_s, AgID_r, performative, content, priority\}$ is a message from agent Ag_s to agent Ag_r with the given performative, content, and priority. $performatives$ are considered to be FIPA-compliant as described in Chapter 2.
- $Bh = \{MendR, \mathbb{RG}\}$ is $\mathcal{A}_i$'s role Behaviour
  - $MendR$ is a means-ends reasoner used to generate plans (see below)
  - $\mathbb{RG}$ is set of goals associated with this role Behaviour
- $\mathbb{B}$ is the set of $\mathcal{A}_i$'s beliefs
- $\mathbb{G}$ is the set of $\mathcal{A}_i$'s goals

- $\mathcal{P} = \{\dots, ab_i, \dots\}$ is the current plan of $\mathcal{A}_i$
- $ab_i = \{\dots, a_{ij}, \dots\}$ is an *action block*, an ordered set of agent actions. Our system have three types of actions:

  Internal actions are executed by agents themselves and are intended to change their beliefs

  External actions are sent to the controller and are executed by it on the environment

  Message actions are used to generate messages to other agents

- *outAcs* is a set of external actions to be executed on the environment
  - $outAcs_i = \{senderID, a^e\}$ is a tuple of sender ID and an external action

The Controller $\mathcal{C}$ is defined as the following tuple:

$$\mathcal{C} = \{\mathcal{I}, inAcs\} \tag{3.3}$$

where $\mathcal{I}$ is an inbox for all agents' outgoing messages and $inAcs$ is the set of all the actions to be applied to the environment.

3.3.1 Agent transition rules

In the previous section we have defined the different elements that compose our framework. Using these definitions we can introduce the operational semantics of our model by means of a set of transition rules. In our model we assume that $\mathcal{M}$ evolves in a set of simulation steps. These transition rules describe how agents' internal states are transformed over each single step of the simulation. They show the part of the state that is changed, the function that changes it, and a summary of the transformation.

The first substep in the agent deliberation cycle is where the agent perceives the current state of the environment. This perception will only modify the agent's beliefs which are then used in the following substeps. Formally, this substep is expressed as follows:

Rule 1: Perceiving the environment

$$\frac{\mathbb{B} \xrightarrow{perceive(E)} \mathbb{B}'}{\mathcal{A}_i \{\cdot, \cdot, \cdot, \mathbb{B}, \cdot, \cdot, \cdot\}, E \rightarrow \mathcal{A}_i \{\cdot, \cdot, \cdot, \mathbb{B}', \cdot, \cdot, \cdot\}, E} \tag{3.4}$$

where *perceive* (...) is a user-defined function that transforms agent $\mathcal{A}$'s set of beliefs based on the environment state E. Doted $\cdot$ elements in the agent tuple are those that are not modified by the perceive function.

During the second substep the agent processes the incoming messages. This is done sequentially, in an arbitrary order. Message processing may modify the agent's beliefs, as well as its goals. This rule is formalized as the following expression:

Rule 2. Message processing:

$$\frac{\mathbb{B}, \mathbb{G}, \mathcal{O} \xrightarrow{process(h)} \mathbb{B}', \mathbb{G}', \mathcal{O}'}{\begin{array}{c}\mathcal{A}_i \{\cdot, \{\{h, t\}, \mathcal{O}\}, \cdot, \mathbb{B}, \mathbb{G}, \cdot, \cdot\}, \cdot \rightarrow \\ \mathcal{A}_i \{\cdot, \{\{t\}, \mathcal{O}'\}, \cdot, \mathbb{B}', \mathbb{G}', \cdot, \cdot\}, \cdot\end{array}} \tag{3.5}$$

where *process* (...) is a user defined message processing function that can modify agent $\mathcal{A}$'s beliefs, goals, and outbox.

In the third substep of the deliberation cycle the agent gets an opportunity to reevaluate and change its goals. The decision is based on its beliefs and current goals, and only the latter is changed during the process. The formal expression for this rule is:

Rule 3. Goal check:

$$\frac{\mathbb{G} \xrightarrow{goal\_check(\mathbb{B},\mathbb{G})} \mathbb{G}'}{\mathcal{A}_i \{\cdot, \cdot, \cdot, \cdot, \mathbb{G}, \cdot, \cdot\}, \cdot \rightarrow \mathcal{A}_i \{\cdot, \cdot, \cdot, \cdot, \mathbb{G}', \cdot, \cdot\}, \cdot} \tag{3.6}$$

where *goal_check* (...) is a user-defined function that may change agent $\mathcal{A}$'s goals based on its beliefs.

One of the most import substeps is the actual reasoning. In this step the agent relies on a means-ends reasoner to generate a plan which it will

follow afterwards. This step is only performed if the current plan has failed or finished.[1] Formally, this step is defined as follows:

Rule 4. Means-ends reasoner:

$$\frac{\mathcal{P} = \varnothing \vee \mathcal{P} = Fail \xrightarrow{MendR(\mathbb{B},\mathbb{G},\mathbb{RG})} \mathcal{P}'}{\mathcal{A}_i \left\{\cdot,\cdot,\cdot,\cdot,\cdot,\mathcal{P},\cdot\right\},\cdot \rightarrow \mathcal{A}_i \left\{\cdot,\cdot,\cdot,\cdot,\cdot,\mathcal{P}',\cdot\right\},\cdot} \tag{3.7}$$

where $MendR\left(\dots\right)$ is some means-ends reasoner that generates plans based on the agent $\mathcal{A}$'s beliefs and goals.

After the plan has been updated, the agent proceeds to execute it. The next action block is extracted from the plan and the set of its actions are run sequentially. This can be expressed with the following formula:

Rule 5. Action execution:

$$\frac{\mathcal{P}\left(h,t\right) \xrightarrow{execute(h)} \mathcal{P}'\left(t\right)}{\begin{array}{c}\mathcal{A}_i \left\{\cdot,\left\{\mathcal{I},\mathcal{O}\right\},\cdot,\mathbb{B},\cdot,\mathcal{P},outAcs\right\},\cdot \rightarrow \\ \mathcal{A}_i \left\{\cdot,\left(\mathcal{I},\mathcal{O}'\right),\cdot,\mathbb{B}',\cdot,\mathcal{P}',outAcs'\right\},\cdot\end{array}} \tag{3.8}$$

where $execute\left(\dots\right)$ is the action execution function that works differently according to the type of action h:

Internal action: *execute* runs h to modify the set of agent $\mathcal{A}$'s beliefs

External action: *execute* appends h to the set of outgoing external actions

Message action: *execute* runs h to generate messages to send

Finally, the agent has an optional choice to reevaluate its role behaviour (i.e. Bh). This substep is similar to the goal check and its result can be either a new role, or no changes. This is defined as follows:

[1] See § 3.4 for more details.

Rule 6. Role check (optional):

$$\frac{Bh \xrightarrow{role\_check(\mathbb{B},\mathbb{G},\mathbb{RG})} Bh'}{\mathcal{A}_i\{\cdot,\cdot,Bh,\cdot,\cdot,\cdot,\cdot\},\cdot \to \mathcal{A}_i\{\cdot,\cdot,Bh',\cdot,\cdot,\cdot,\cdot\},\cdot} \tag{3.9}$$

where $role\_check(\dots)$ is a user-defined function that may modify agent $\mathcal{A}$'s role (behaviour) (and, consequently, role goals) based on its beliefs and goals.

3.3.2 MAS transition rules

After formally defining the internal deliberation process of agents in our framework, in this subsection we will describe the Controller's execution of the environment on each simulation step.

First of all, there is an optional substep during which the controller can modify the environment before agents' actions get applied to it. Formally, we can express it in the following way:

Rule 1. Pre-action execution step (optional):

$$\frac{E \xrightarrow{pre\_step()} E'}{\mathcal{M}\{E,\cdot,\cdot\} \to \mathcal{M}\{E',\cdot,\cdot\}} \tag{3.10}$$

where $pre\_step()$ is a user-defined function that modifies the environment before the agents' actions are collectively applied to it.

After that, controller acts as a message dispatcher, processing all the outgoing messages and passing them to the corresponding recipients in a sequential manner. This is defined as follows:

Rule 2. Message forwarding.

$$\frac{\begin{array}{c}\mathcal{A}_r\{\cdot,\{\{x\},\mathcal{O}\},\cdot,\cdot,\cdot,\cdot,\cdot\} \xrightarrow{fwd\_msg(h,\mathcal{A}_r)} \\ \mathcal{A}'_r\{\cdot,\{\{x,h\},\mathcal{O}\},\cdot,\cdot,\cdot,\cdot,\cdot\}\end{array}}{\begin{array}{c}\mathcal{M}\{\cdot,\{\dots,\mathcal{A}_r,\dots\},\{\{h,t\},\cdot\}\} \to \\ \mathcal{M}\{\cdot,\{\dots,\mathcal{A}'_r,\dots\},\{\{t\},\cdot\}\}\end{array}} \tag{3.11}$$

where $fwd_msh(\dots)$ is a simple function that moves the specified message from sender's inbox, to recipient's outbox.

The next substep executes all the action sent by agents. This process is also done sequentially. This substep is defined by the following expression:

Rule 3. Action execution:

$$\frac{E \xrightarrow{execute\_ac(a)} E'}{\mathcal{M}\{E, \cdot, \{\cdot, \{a, t\}\}\} \rightarrow \mathcal{M}\{E', \cdot, \{\cdot, \{t\}\}\}} \tag{3.12}$$

where $execute\_ac(\dots)$ is a function that applies the action h to the environment.

Finally, there is another optional substep to modify the environment. It is similar to the first substep, differing only in the used function. Formally:

Rule 4. Post-action execution step (optional)

$$\frac{E \xrightarrow{post\_step()} E'}{\mathcal{M}\{E, \cdot, \cdot\} \rightarrow \mathcal{M}\{E', \cdot, \cdot\}} \tag{3.13}$$

where $post\_step()$ is a user-defined function that modifies the environment after the agents' actions are collectively applied to it.

3.4 Means-ends reasoning: Hierarchical Task Networks planner

In our base model we chose not to specify any fixed reasoning model ($MendR(\mathbb{B}, \mathbb{G}, \mathbb{RG})$ in agent's **Rule 4**) as the sole reasoning mechanism, leaving it up to user to select one (and providing the required specifications for it).

However, for our basic implementation version of the system we have selected an HTN planner [31][32] as an instantiation of the $MendR$ function, as it was already shown[2] that HTN planners can fit well in BDI reasoning process [60].

The basic HTN is a tree composed of Abstract and Primitive Tasks. Primitive tasks are leaves of the tree, while other vertices are abstract tasks. The former are actual actions to be added to a plan and represent a decompositions

[2] See § 2.1.1.1.

of compound abstract tasks while the latter act as a glue, creating the structure that goes from the most general tasks at the top, to the basic, primitive ones. Each abstract task usually has some precondition that helps to reduce the actual part of the tree to be traversed during the planning and may have different policies on how to treat their children during planning (we can add the first one with satisfied preconditions, all of the, etc.).

There exists a variety of HTN-based models, with different representation and planning procedures. As most of these models deal only with high-level planning, distancing from the practical planning and following task execution, we have considered choosing HTN planner models used for actual games. These models have been adapted not only for effective abstract-to-concrete facts mapping, but also for replanning during execution. As a result, we have selected [40] as a basis for our work, and slightly modified it to better suit our needs and COMPSs restrictions. These modifications covered for the difference in the perception models (the original algorithm used active sensors), and allowed the partial sets of method's subtasks to be accepted as parts of the plan, for the greater flexibility of the model (as described in the next paragraph).

In this model the abstract tasks are divided into two task types: Compound Tasks and Methods. Compound task only consists of an ordered set of Methods while each Method consists of an ordered set of Compound or Primitive Tasks, conditions and some additional information. Primitive tasks contain action blocks to be executed, consisting of action of various types, and also some preconditions.

During the planning the following rules apply:

1. The root is a compound task

2. When processing a compound task, we select for further decomposition the first method which conditions are satisfied (the other methods of this compound tasks are discarded)

3. If all methods' conditions are not satisfied, the compound tasks fails and we roll back

4. When processing a method, we add to the plan all of its primitive tasks for which preconditions are satisfied

5. When adding a primitive task to the plan, what we actually add is its action block

6. In a method we may specify that all the satisfied primitive tasks' actions of this method must be merged into a single action block

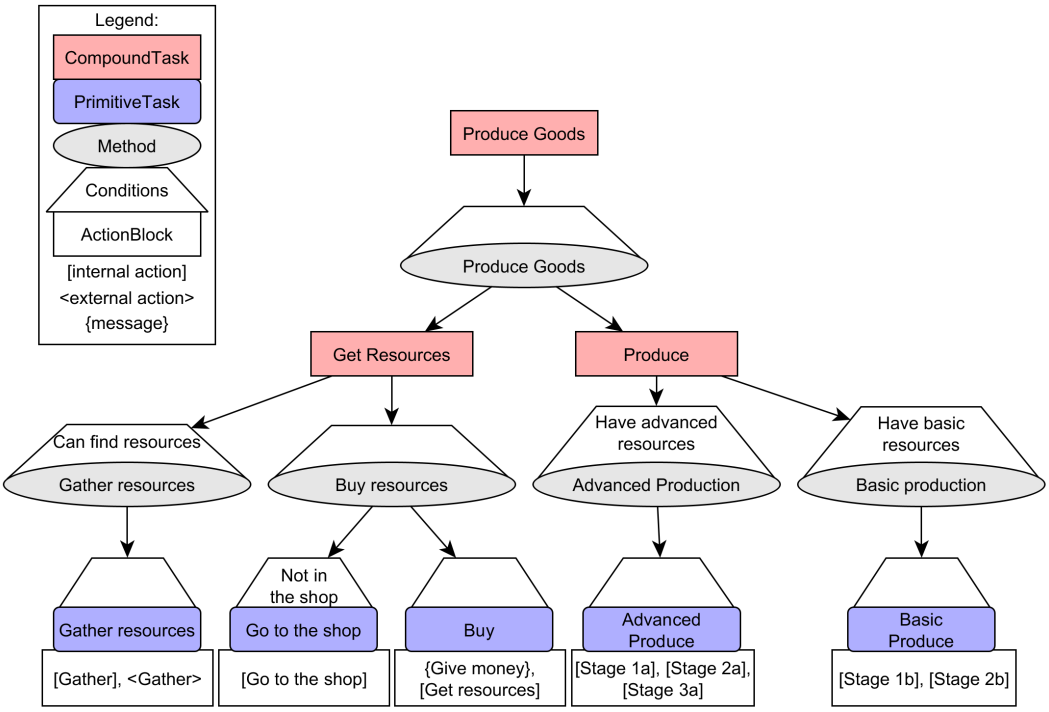

Figure 3.5: HTN example

Fig.3.5 shows an example of an HTN that follows this structure.

In the case of HTN planner, we may consider the current root of HTN as an agent's goal. By following this simple rules, we can easily obtain a sequence of actions to be executed based on the current beliefs and goals.

In the current version of our HTN model after the current plan has finished we generate a new one, as per **rule 4**. Thus, the only explicit type of goals we support now is maintain goal [26], but other types of goals (for example, achieve goals or perform goals) can be simulated by the proper use of *goal_check* function. And in the case of HTN as means-ends reasoner, *goal_check* () can easily switch between different compound tasks in HTN (or select a new HTN) as a method of goal revision. This can be done either by traversing the graph of HTN, or by generating a new HTN and replacing the old one with it (though one should always be careful about the belief sets and ontologies used by each planner).

3.5 Chapter summary

In this chapter we have introduced the COMPSs framework and described the main principles of its design and workflow, including its ease of use (simple modifications to the sequential code), and versatility (can be deployed to a vast range of systems, directly and via containers). We have covered how it uses code and data flow analysis along with several code annotations to turn sequential programs into distributed ones. Also we have described our discrete-step agent-based simulation model. We have proposed formal definitions of the framework's components and have introduced 6 transition rules that guide agents during the simulation and 4 rules that are used by the controller to forward messages and run the environment. Finally, we have described the HTN model used in our framework as a basic implementation of $MeanR$ reasoning function.

In the following chapter the implementation of the system is described along with the relevant technical details and usage guidelines.

Chapter 4

Implementation

In the previous chapter we have introduced the agent-based simulation model and described its design features.

This chapter will describe an instantiation of the frameworks in a concrete implementation. First, we will introduce the implementation of the main execution cycle, user API, and the main object classes in section 4.1, and then move on to cover the implemented reasoning tools in section 4.2.

4.1 Multi-agent system

The whole system is implemented in Python, supporting both Python 2 and Python 3. It is organised as a package with several submodules and a list of main classes that are easily accessible by the user.

These modules are:

Controller : it contains the `Controller` class, the main utility entity in the system, responsible for containing the MAS, running PyCOMPSs tasks, forwarding messages, and executing all the utility methods

Agent : this module contains the `Agent` class that is a simple container for the agent description.

Behavior : it contains the `Behavior` class, that implicitly for the user provides full agent's workflow; user might need to inherit their own classes from it for complex behaviours

Directory : it contains `Directory` and `DirectoryEntry` classes that are used by the built-in directory facilitator

HTN : contains all the HTN-related classes, as well as some of the more general ones; the full list is: `CompoundTask`, `Method`, `PrimitiveTask`, `Effect`, `Action`, `ActionBlock`, `HTNPlanner`, `BeliefSet` and `Conditions`

Messaging : this module contains `Message` and `Messagebox` classes used for agent messaging

State : this module contains a simple `State` structure, used to transfer and work with persistent states.

Of all of these classes, users have direct access to (and should work directly only with): `Controller`, `BeliefSet`, `Behavior`, `ActionBlock`, `Conditions`, `CompoundTask`, `Method`, `PrimitiveTask` and `Effect`. These classes are described in following sections, along the instructions on how to use them.

The simulation structure is defined in the Controller class and follows the scheme shown in Fig.4.1. Essentially, a simulation run for a predefined number of steps; during each step each agent's behavior step is marked as a PyCOMPSs task and distributed over the computation environment. After the computations are done, their results, in form of lists of outgoing messages and actions, are collected by the controller. Messages get forwarded to their recipients, actions get applied to the environment, and the iteration step ends, starting the next one.

This model of running the simulation was chosen on purpose to ensure the uniform flow of discrete simulated time for each agent.

Implementing and running a very basic MAS with this package is quite simple. First of all, the user needs to create a Controller. There is a number of options for Controller initialisation, but the simplest one does not use any parameters:

```
1 from Library import Controller
2 controller = Controller()
```

In order to populate the system with agents, they need to be generated and registered in the Controller. This can be done by using the Controller's `generate_agent` method. Once again, there is a lot of possible options, but the most basic one requires just the name specification (it may not be unique, each agent is assigned a unique identifier on generation):

```
1 controller.generate\_agent("MyAgent")
```

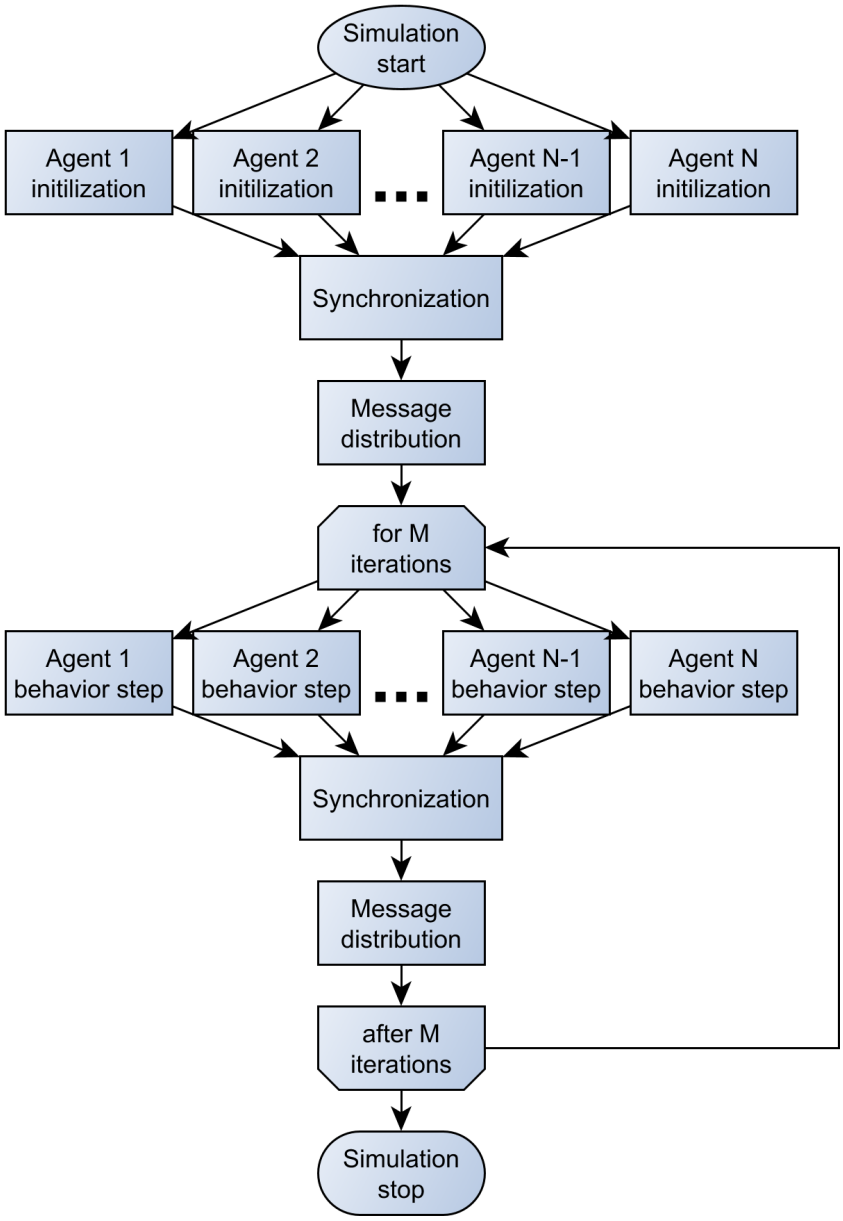

Figure 4.1: The simulation structure

Note, however, that this is an empty agent: it does not contain any behavior or reasoner by default.

In order to run a simulation, the user must use the Controller's `run` method. It takes, as input, only the requested number of iterations. For instance:

```
controller.run(10)
```

will run the simulation for 10 simulation steps.

However, as it was mentioned, it is an empty simulation with each agent doing nothing during its step. In next subsections, each package module is detailed and what can be accomplished with them is shown. Also, the agent simulation cycle is described.

4.1.1 Behavior class

The Behavior class acts as a basic template for specifying agents' behavior. The class itself is essentially a wrapper for several important functions: `perceive`, `process`, `goal_check`, `reason`, `execute`, and `role_check`[1]. It also contains several auxiliary fields only used for temporary storage of values; they are reset at each step.

The main method in the Behavior class is `step`. It follows the structure defined by the transition rules (see Fig.4.2).

At the beginning of an agent's step, its behavior retrieves the agent's state, composed of its beliefs and the planner state. The agent's goals $\mathbb{G}$ are represented by the state of the planner (namely, its current root) and the agent's role goals $\mathbb{RG}$ by the selected behavior itself, as different behaviors potentially imply different implementations of various transition functions.

First of all, the agent perceives the environment. A copy of the environment is passed to the behavior's `perceive` method along with its beliefs, and the modified beliefs are read from its output:

```
state.beliefs = self.perceive(environment, state.beliefs)
```

This function can be redefined by the user in a subclass. By default, it ignores the environment and merely returns the agent's current beliefs. The way to modify beliefs is covered in the corresponding subsection, though it can be mentioned here that you can work with the BeliefSet class quite similarly to the standard Python dictionary class (its structure is based on it).

The next stage is message processing. The `process` function is called for each message in the agent's inbox:

```
for message in inbox:
```

[1] These function correspond to the functions in transition rules described in § 3.3.1

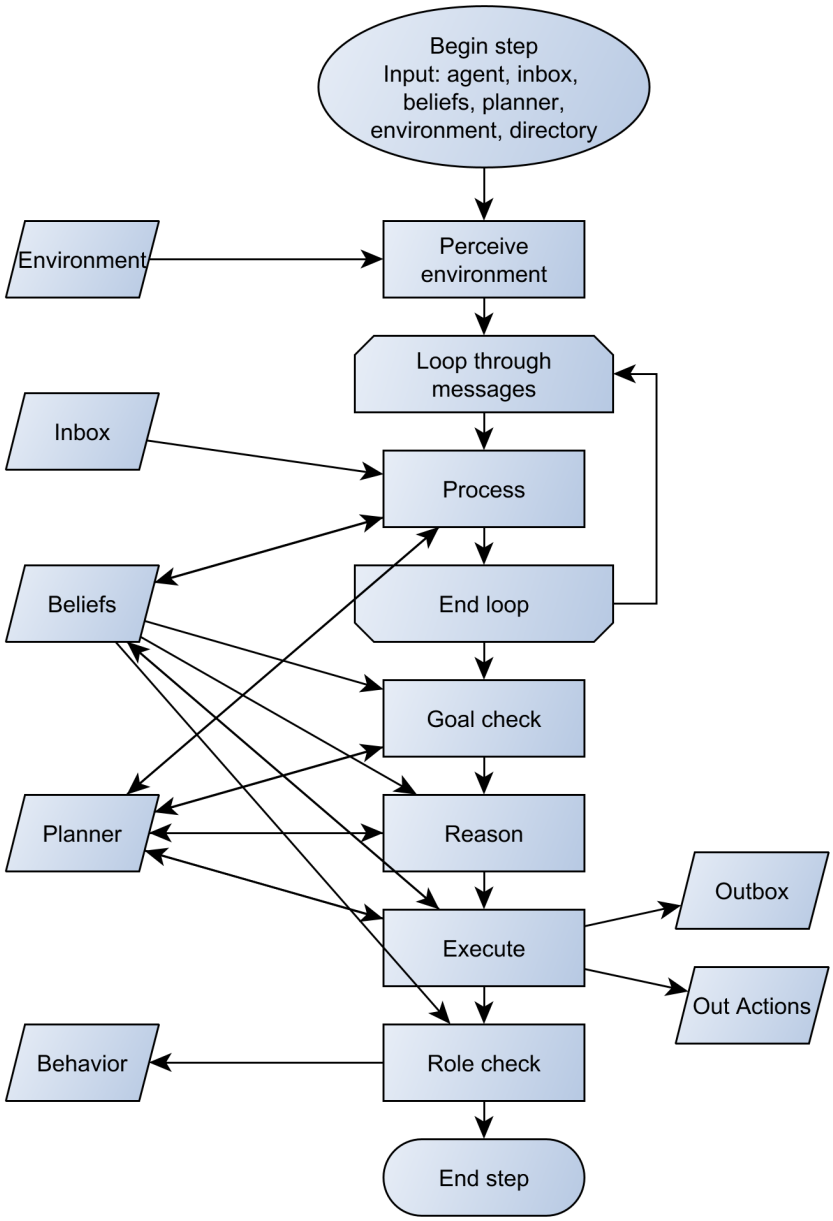

Figure 4.2: Agent behavior step structure

```
2     state.beliefs, state.planner, reply = self.process(message,
          state.beliefs, state.planner)
```

This function should be redefined by the user to specify possible reactions to incoming messages. Its inputs are: the message being processed, the agent's

current beliefs, and the planner. It must return modified beliefs, a modified planner and a reply (the latter can be set to `None`if there is no reply). Possible ways to modify the planner will be covered later in this subsection. If the agent needs to send a reply to the message, it can use the `Message`'s built-in method `reply` and return its output along with the modified beliefs and planner.

After that, and following the Rule 3, the goal check is performed. Once again, the user can override this function if they might need to switch goals, otherwise the default identity function is used.

This function is applied in the following way:

```
1 state.planner = self.goal_check(state.beliefs, state.planner)
```

Thus, it retrieves the agent's beliefs and the planner and returns the modified version of the latter.

Following the goal revision, the reasoning part of agent's execution is done. Here the user has two options:

1. If the planner was passed as an argument to the Controller during the agent's generation, the HTN planner will be used.

2. Otherwise, the behavior will look for a default action block that can also be passed at the agent generation (the empty action block is used by default).

The default action block option is simple: the behavior copies the actions in it and considers them, as if they were its plan. Otherwise, it depends on the state of the planner. If the plan has finished, or failed, the HTN will try to re-plan; otherwise, this step is skipped.

After reasoning, the agent acts upon its current plan, no matter how it was achieved. The `execute` function is called in the following way:

```
1 state.beliefs = self.execute(agent, block, state.beliefs,
    directory)
```

The processing of each action depends on its type and is detailed in the next subsections.

Finally, after the execution is finished, the agent may review its role, i.e. behavior. A user-defined `role_check` function is used for that purpose:

```
1 role = self.role_check(state.beliefs)
```

By default, it always returns `None`. However, if the redefined version of it returns an instance of some Behavior subclass, the corresponding request of Behavior change will be composed for the controller, and it will get changed before the next iteration of the simulation.

There are also 2 special fields accessible via the `self` parameter: `self.log` and `self.finished`. The first one contains the string log to which agents may write anything. The log gets flushed to the controller after each step and is written to a file if the user has specified one.

`self.finished` is related to a special experimental type of execution: it is possible to make each simulation step span for several execution steps. In this case, each simulation step lasts for either a specified number of execution steps, or until all the agents set their `self.finished` flags to `True` (finished agents are excluded from the further execution in the same simulation step to optimise the load).

In the following subsections we will cover the three available types of actions.

4.1.1.1 Internal actions

The most simple action to process. A simple one line of code is used to deal with it:

```
1 beliefs = action.content(agent, beliefs, *action.args, **action
    .kwargs)
```

The agent object is passed in case the action requires some of the agent's descriptions. `*args` and `**kwargs` are additional parameters the action function may require. These parameters are used to allow for generalised functions to be passed as action functions. For example, if one agent needs an internal action that increases belief A by 2, and the other agent needs the same action, but with an increment of 10, we may create a function that increases this belief by N using this parameter's structure and specify the N for each concrete action.

Besides just beliefs, the user can also return a "finished" flag (for the special case where the user defines the simulation step to span for several execution steps), and a log string that the controller will print if the user has specified a log output file.

4.1.1.2 Message action

In this case, the requirements for the action function are different, as well as its expected outputs:

```
msgs = action.content(agent, beliefs, directory, *action.args,
    **action.kwargs)
                for msg in msgs:
                    r, p, c, priority = msg
                    self.outbox.put(Message(sender=agent.id,
                        receiver=r, performative=p, content=c,
                        priority=priority))
```

So the action function for message actions should accept as parameters: the agent object, its beliefs, the directory of agents, and optional `*args` and `**kwargs` arguments. It must return an iterable collection of tuples of the following format: `receiverID, performative, content`.

One of the important functions of messaging is the possibility to send requests to the controller. In order to do this, the receiver should be stated as `"controller"` instead of the agent id, the performative should contain the request, and the content should contain the parameters of the request. Here is the full list of currently supported requests:

> **`"finished"`** , no arguments: mark the agent as finished and exclude it from the simulation.
>
> **`"print"`** , {`"print": object_to_print`}: request the controller to print the sent object in the standard output.
>
> **`"stop"`** , no arguments: request to preemptively stop the simulation run.
>
> **`"agent"`** , possible argument keys: name, behavior, beliefs, services, default_block, planner: request generation and registration of a new agent with specified parameters (details on these can be found in the Controller subsection).

For messages actions, the user can also return a log string after the list of messages.

4.1.1.3 External action

Finally, the external action is a bit different from the internal and message actions. External actions are not run during the agent's execution; instead,

they are appended to the queue of outgoing actions for the controller. Thus, they will be run by the controller itself in the following manner:

```
1 self._environment = request["function"](self._environment, *
    args, **kwargs)
```

As it can be seen, it takes an environment and additional arguments as input and returns a modified version of the environment.

As an additional feature, the user, during the action definition (see below) of an external action, can also specify a parameter called `beliefs` (an iterable collection). In this case, before the action is added to the outgoing action queue, the corresponding beliefs are added with their values to the `kwargs`. This is a way to pass a belief-dependent action to the environment.

Besides the aforementioned step method, there is also an additional notable one: `initialisation`. This method is run once for each agent at the very beginning of the simulation. It requires an `init_block` to be defined at the agent generation (otherwise it defaults to an empty block). These `init_block`s are run by this method in the same way the execution function runs the plans.

4.1.2 Controller class

The Controller is the most complex class of the whole system, as it maintains the simulation and offers a range of utility functions.

The full initialiser for the Controller has the following structure:

```
1 def __init__(self, environment=None, prestep=None, poststep=
    None, msg_handler=None, step_limit=1, log_file=None,
    verbose=False):
```

During the controller initialisation, the user can specify a list of parameters: an environment, environmental pre- and poststep functions, a message handler, a step limit, a log file, and the verbosity.

The environment and the pre- and poststep functions are exactly the same as in the formalisation in § 3.3.2. There are two relevant insights about the environment structure: first, it can be an instance of any class, but it must have a `copy()` method that returns its copy to be sent to agents; and second, the environment variable **must** be serialisable by the Python's `cPickle` package[2]. Therefore, a simple dictionary works considerably well as environment.

[2]Due to the way PyCOMPSs serialises objects, the user should be vigilant in order to ensure their custom objects are serialisable before using them inside of the framework (for

`log_file` is as described in the previous sections: it specifies a file, to which the execution logs will be printed whenever there are any. `step_limit` is the upper bound number of execution steps for each simulation step (see the previous subsection). Finally, `msg_handler` is the special function that adds customisable processing of the outgoing messages. All these messages pass through this function (identity function by default) before getting forwarded.

Though there is a lot of attributes inside the Controller class, most of them are hidden (prefixed with "_").

The few public functions are setters for the environment, the pre-step and post-step, and verbosity, `get_count()` returning the current number of agents in the simulation, and three large methods: `generate_agent`, `register_agent`, and `run`.

The main function among them is the `run` method that starts the simulation and controls the execution loop:

```
1 run(num\_iter, performance=False):
```

It runs the simulation for `num_iter` steps. If the `performance` parameter is set to True, then this method will also return the performance data as a tuple formatted as `(agent_time, controller_time`. The former is the total time of the simulation that was spent to compute the behavior steps, and the latter is the time spent by controller's utility functions.

Another important function that the controller provides is the agent generation. By using it, the user does not need to explicitly deal with a number of various inner classes; rather the user just needs to pass the required parameters and they will be set and assigned automatically.

```
1 generate_agent(self, name, behavior=None, beliefs=None,
    init_block=None, planner=None, default_block=None, services
    =None, register=True):
```

This method generates an agent and returns it. Besides the aforementioned name parameter, the user may specify a custom behavior for this agent, its starting beliefs, the initialisation action block, the planner or default action block (the planner will always take precedence), the list of the string service names of this agent for the directory facilitator, and an important boolean `register` denoting whether the generated agent should also be registered in

instance, as the beliefs or the environment). The Python's `cPickle` module is used as a serialiser; it does its job several orders of magnitude faster but with more strict constraints. For instance, non-top-level functions are forbidden as object field values, including lambda-functions.

the controller. Most of these parameters are actually required for the registration (that is a default strategy for this method). For pure agent generation without registering it, we would only need a name and, possibly, a behavior.

```
1 register(self, agent, beliefs=None, init_block=None, planner=
    None, default_block=None, services=None):
```

All of the parameters, except `agent`, of this method are also optional parameters for the `generate_agent` because of the option to simultaneously generate and register agents.

4.1.2.1 PyCOMPSs tasks

PyCOMPSs tasks form another notable part of the Controller module and, thus, they deserve a closer look.

There are two PyCOMPSs tasks present: `agent_step` and `agent_init_step`, the latter being a slight alteration of the former, replacing the behavior's step method by the initialisation method.

`agent_step` is a PyCOMPSs wrapper for the execution of a single agent. It receives as parameters: the corresponding agent object, its inbox, the directory of agents, and the environment. Note that there is no state in the parameters. The reason is that as the agents are highly and inconsistently distributed, we decided to follow the paradigm of the stateless agents, i.e. their states are detached from them at the end of every step, and sent back at the beginning of the next. This is done via the messaging mechanism. That is where the agent's state is located in the inputs[3].

The rest of the task structure is straightforward: as precaution the behavior is sterilised of possible remnants of previous temporary data, the state is extracted from the agent's inbox, the agent behavior's step method is invoked, the outgoing messages and actions are collected along with the renewed state, the behavior is cleared again, and all the outgoing messages and actions are returned.

4.1.3 Agent class

The Agent class is a simple container for a few values and the corresponding agent's behavior. It also provides a couple of utility functions for message composition based on the ID of this agent.

The Agent's fields are:

[3]In the future work we will explore the option of using `dataClay` framework to provide state persistence to agents. See § 6.3

id : unique identifier of the agent

name : string name of the agent

behavior : the current behavior of this agent

4.1.4 BeliefSet class

BeliefSet is a custom class for containing various beliefs and their values. It provides some additional functions in conjunction with other classes from the HTN module. The basic functionality of BeliefSet if similar to the one of the standard Python dictionary type: beliefs are set in exactly the same way. Still, there are two restrictions compared to dictionaries. The first one is that the keys must be unique strings. The second one is that sets and tuples cannot be used as values due to the special ways BeliefSets are dealt with by the HTN planner. The custom class instances are possible but, in order to be used properly, they are required to have a default equality checker implemented, inequality checkers, and a hash function.

4.1.5 Action and ActionBlock classes

Action is a rather simple container for action functions. It contains the following fields:

type : the string of the action type; either `"internal"`, `"external"`, or `"message"`.

content : the action function. **IMPORTANT NOTE:** due to the way PyCOMPSs serialises custom objects, only top-level functions are allowed to be used as values for object fields (like Action's content).

name : the name of the action for logging.

beliefs : beliefs for external actions, as described above

args, kwargs : additional parameters of the actions.

ActionBlocks are iterable containers for lists of Actions. Besides action, they also contain the `parent` field that contains a pointer to the method that is the parent of the primitive tasks containing this block in HTN. This is needed for block grouping during the planning.

ActionBlocks contain three important methods that users should use, designed to add actions to the ActionBlock without invoking the Action class directly:

```
1 def add_internal_action(self, action, name="<action>", *args,
      **kwargs):
2     ...
3
4 def add_external_action(self, action, name="<action>", beliefs=
      None, *args, **kwargs):
5     ...
6
7 def add_message(self, action, name="<action>", *args, **kwargs)
      :
8     ...
```

4.1.6 Directory class

The instances of this class act as iterable containers of the information about agents. `Directory` contains two fields for that: white pages, which is a dictionary from agent IDs to the corresponding agent entries, and yellow pages, which is a dictionary mapping all present service names to the agent entries of the agents that provide them.

Agent entries consist of the basic information to contact the agents: ID, name, and services list.

4.1.7 Message class

Although the user does not directly create Messages, they can process them in the `process` method. The Message is a simple class, consisting of the following main fields:

`sender` : ID of the message sender

`receiver` : ID of the message receiver

`performative` : string of the message's performative. The proposed FIPA ACL performatives to use are[4]:

propose: propose to do some action

[4]The full list of FIPA ACL performatives are presented in the FIPA Communicative Act Library Specification (SC00037J) [3]

accept-proposal: agree to a proposal

reject-proposal: reject a proposal

inform: provide some information

query-ff: ask about some information

request: ask to perform some action

agree: agree to do some action

refuse: refuse to do some action

`content` : the content of the message, usually, a dictionary

`priority` : a boolean; if true, during the message processing such messages would be inserted at the beginning of the inbox, not appended to its end.

4.2 HTN planner

The planner is implemented following the basic version of the planner from [40], as explained in 3.4.

The main classes are defined in the package's `HTN` module. These classes are: `BeliefSet`, `Conditions`, `PrimitiveTask`, `Effect`, `Method`, `CompoundTask`, and `HTNPlanner`.

4.2.1 BeliefSet and Conditions

`BeliefSet` has been covered in subsection 4.1.4.

The `Conditions` class is derived from `BeliefSet` and modifies it in a number of ways. First of all, it can check against a `BeliefSet` object to see whether the conditions are a subset of its belief set (thus, all conditions are satisfied): `check_conditions(, beliefs)`. Secondly, it also accepts `tuple`s (pairs should be used, other elements will be ignored) and `set`s as special values for its beliefs. If a belief is defined as a `tuple`, the `check_conditions` function will check whether the corresponding `BeliefSet`'s belief value lies between the first and the second elements of this tuple. In case of the `set`, it checks whether the belief's value is equal to one of the values in the list.

4.2.2 Tasks

`CompoundTask`s are intermediate elements in the task networks. They do not have any actual code to be executed during the runtime, but rather contain a list of `Methods` denoting the way this task can be completed, and a parent field containing a pointer to its parent method for the HTN navigation purposes.

Each `Method` contains a `Conditions` object, a `list` of subtasks (that should be either compound or primitive), and the `parent` field. When a planner checks a method, it tests its conditions against the planner's current belief set to see whether this method is applicable. The methods are checked in the order of the subtasks list. As the order is important, it is possible to specify where a new task should be inserted.

`PrimitiveTask`s are the containers for the ActionBlocks agents will try to perform. They have a field for ActionBlock (`action_block`, defined during the initialisation of the object and also `preconditions` and `effects` properties.

It is important to note, about the effects field of `PrimitiveTask`, that effects are quite similar to post-conditions. They are used by planners to "simulate" the changes to beliefs after using this `PrimitiveTask`. This is necessary for the sake of further planning.

`effects` is a list of instances of the `Effect` class. Each `Effect` contains a belief name and either a string denoting the operation that should be applied to the value of this belief and the modifier value if it is used, or a function that will be applied to this belief, to simulate more complex effects. Once again, please note that although using lambda expressions as `Effect`'s effect would have been a sound solution, COMPSs framework does not support serialisation of non-top level functions (as it is using the `cPickle` module for serialisation).

Also please note that `effects` should act exactly in the same way as actual `PrimitiveTasks`, especially in the case of heavy belief inter-dependencies. They just need to be consistent with conditions used for Methods and the PrimitiveTasks scheme. This way, planners may not get plans as long as they could have had, and may end up with only partial plans but, after these partial plans complete, re-planning will continue the process using the actual states of the belief sets. It is important to remember that this step is just a "simulation" of the plan execution. It should be faster than an actual run of the plan. It is entirely up to the user to find the balance between the complexity of the planning effects simulation and the plans length.

4.2.3 Planner

As it was explained in Section 3.4, the `HTNPlanner` is based on the hierarchy of tasks. It starts with a root compound task and a copy of the current set of beliefs and expands via `CompoundTasks`' `Methods` and `Methods`' subtasks. During an iteration of reasoning, a planner traverses the graph once, following a depth-first policy. During the search, it maintains a stack of tasks to process. When the planner processes a `CompoundTask`, it checks the conditions of its methods one by one in the order defined by the user during the methods insertion. The first `Method` whose conditions are satisfied by the current beliefs is selected. If no method is applicable, the planner rolls back to the previous unprocessed task undoing the changes to the current plan if there were any (if it was checking the root, it returns the current plan). The selected method's subtasks list is added to the top of the stack of tasks to process (in the original order).

When the planner processes the `PrimitiveTaks`, it also checks it preconditions. If they are satisfied, this `PrimitiveTask` is added to the current plan and its `Effects` are applied in order to the current version of the simulated belief set. If they are not satisfied, the planner rolls back.

4.2.4 Implementing custom HTN

To make use of the HTN planner, users need to define the HTN and create an `HTNPlanner` object based on it. Some examples of the HTNs and their representation in the form of code is given in the appendix A.

`CompoundTask`'s constructor:

```
1 def __init__(self, methods=None, name=""):
```

It is possible to pass no arguments during the initialisation and insert (or remove) the methods later via `add_method(value, index=None)` and `remove_method(index)` methods. Name is an optional field that can be used to display the understandable task name in the verbose mode (note that as planning happens in the corresponding task nodes in COMPSs, this planner runtime description is outputted to the corresponding job's `.out` files).

`Method`'s constructor

```
1 def __init__(self, conditions, subtasks=None, append=False,
    name=""):
```

`Conditions` must be passed as an argument during the initialisation. Subtasks may be added later in the same way the methods may be added to `CompoundTasks`: `add_subtask(value, index=None)`, `remove_subtasks(index)`.

`PrimitiveTask`'s constructor:

```
1 def __init__(self, name, preconditions, effects=None,
    action_block=None):
```

`name` is a string used in the verbose mode.

`preconditions` is a `Conditions` object.

`effects` is a list of Effects

`action_block` is the block of action corresponding to the task.

`Effect`'s constructor:

```
1 def __init__(self, belief=None, effect="add", value=None):
```

`belief` is the name of the belief to modify.

`effect` is either a string denoting the operation to be performed or the function to be applied; the function should accept beliefs as an input and return modified beliefs as an output.

`value` is the modifier's value (if applicable).

The list of the supported `effect` actions and the corresponding operations:

`add` : addition

`sub` : subtraction

`mul` : multiplication

`div` : division

`mod` : modulus

`pow` : power

`rep` : reassignment

`and` : logical AND

`or` : logical OR

`not` : logical NOT

`BeliefSet`'s and `Conditions`'s constructors:

```
1 def __init__(self, beliefs=None):
```

`beliefs` must be passed as a dictionary. If nothing is passed, an empty dictionary is created.

`HTNPlanner`'s constructor:

```
1 def __init__(self, root_task, verbose=False):
```

The user just needs to pass the top level compound task (as `root_task`) as all the network is reachable from it. If `verbose` is `True`, the planner will generate a short summary of the current step and, in case of re-planning, the jobs will also contain the trace of this process.

Re-planning is made by invoking the `planner.replan(beliefs)` method. It clears the current plan, and starts the breadth-first decomposition of the HTN, beginning by the root compound task. Each method of this compound task is checked and the first one the conditions of which are satisfied is selected. The sub-tasks of the selected method are added to the queue of tasks to check. When an element is dequeued, what happens next depends on its type: if it is a compound task, the procedure is the same as with the root task; otherwise, if it is a primitive task, its preconditions are checked and if they are satisfied, this primitive task's action block is added to the plan. As an additional feature, it is possible to set `method.append` to `True`. In this case, the action blocks of all the child tasks of this method (to the very bottom of the HTN tree) will be put together in one action block.

The planner automatically follows the generated plan block by block. All that agents need to do is to call `planner.next_block()` to get the next action block of the plan (this is done once per `agent_step` as specified in the Behavior description).

4.3 Limitations and possible improvements

As the framework implementation is still a raw prototype, various things may be improved from the implementation point of view.

First of all, a sophisticated logging system is needed. As of now, the user is responsible of either directly writing to files from the external action and pre- or post-step functions. A universal logging system based on the current log objects may be implemented and optimised.

Besides the logs, the framework might benefit from additional goal types, thus enhancing the capabilities of the planner and the possible complexity of the simulation.

Although now the environment can be assigned as any object, it makes sense from the formalisation and data persistence point of view to create a predefined structure for the purpose. Moreover that will allow us to introduce the concept of agent scopes: as the environment can be huge and not fully observable, it makes sense to send to agents, or provide them, only the specified part of it.

4.4 Chapter summary

In this technical chapter, we have gone through the details of our system's implementation. We have described its various components and classes, including the Behavior's step function, that implements agent's transition rules; covered the Controller objects' structure and the interactions with the PyCOMPSs package. We have also provided the overview of HTN planner and its simple BFS-based task enumeration strategy.

In the next chapter, we will provide the information about the experiments we have performed to showcase this system's performance and their results.

Chapter 5

Experiments

In the previous chapter we have provided the implementation details of the proposed model. We have specified the runtime execution of the simulation cycle and the agent deliberation cycle. Also we have provided the descriptions for the most important functions and methods of the system. Finally, we have introduced our version of HTN planner that is provided as the default means-ends reasoner in the framework.

Now we will demonstrate how our system works from the point of view of some experiments. First, we will describe these experiments which are designed to showcase the performance metrics of the system. Then we will introduce a real-world scenario and show how it can be implemented within our framework.

There are two sections in this chapter: in section 5.1 we outline the structure of our experiments and in section 5.2 we provide their results with respect to specified criteria.

5.1 Structure of the experiments

In order to test the system's performance and functionality, we have designed a number of experiment sets to test different aspect of the system:

1. The first set of experiments is to demonstrate the basic functionality and correctness of implementation. They demonstrate that the system runs as expected: no messages lost, no agent gets stuck for unreasonably long time in the computations and the simulation loop runs in an orderly manner for the specified number of steps.

2. With the second set of experiments we focus on testing the limits of the

system's performance. For that we propose three numeric aspects of the simulation as parameters: the total number of agents, the total number of messages, and the size of each message, with the performance time being the metric.

3. And the third set of experiments is designed to showcase the reasoning model, its expressiveness, effectiveness, and, once again, performance. These experiments focus on implementation of a real-world scenario. The most important criteria in this case is empirical: we need to ensure that the agents in the system act similarly to what would be expected of the corresponding actor in the reality.

5.1.1 Functionality tests

In order to verify that during the runtime the system works as intended, we have designed three simple tests scenarios: `incrementation`, `ping`, `random messaging`, which involve different components of the system.

5.1.1.1 `incrementation` scenario

The scenario can be summarised as follows:

- There are 3 basic agents in the simulation.
- Each of them uses only a simple default action block
- These default blocks contain only one internal action that increments the value of the agent's `counter` field

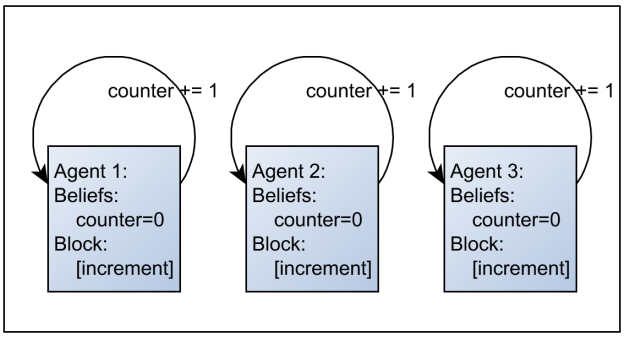

Figure 5.1: Increment scenario scheme

Formally, this simulation is not a MAS, as the agents do not interact, but it allows us to easily look into agents' inboxes as they are of manageable size and there are only three of them.

We expect:

- The simulation to last for exactly the specified number of round.
- The counter to increase uniformly and be equal to the total number of simulation steps at the end of the run.
- Inbox/outbox to contain only the state of the agent.

5.1.1.2 `ping` scenario

This scenario is described the following way:

- There are 3 agents, the reasoning mechanism of each one based on an HTN planner
- The HTN used is shown on Fig.5.2
- A simple environment is used with a single field: `counter`
- Agents' beliefs are more complex to cover the whole range of HTN interaction; they consist of:
  - boolean `got_msgs`, denotes whether the agent has received a message
  - integer `reply_count`, denotes how many replies the agent has to send (note that for the testing purpose, the reply output of the `process` method is not used in this scenario)
  - list `message`, denotes the messages the agent has received
  - integer `counter`, used to count the number of "pong" messages the agent has received
  - integer `to_env`, denotes the value to be added to the environment's `counter`.

  .

During the simulation, each agents acts in the following way:

- If the agent received a message, it processes it depending on its content:

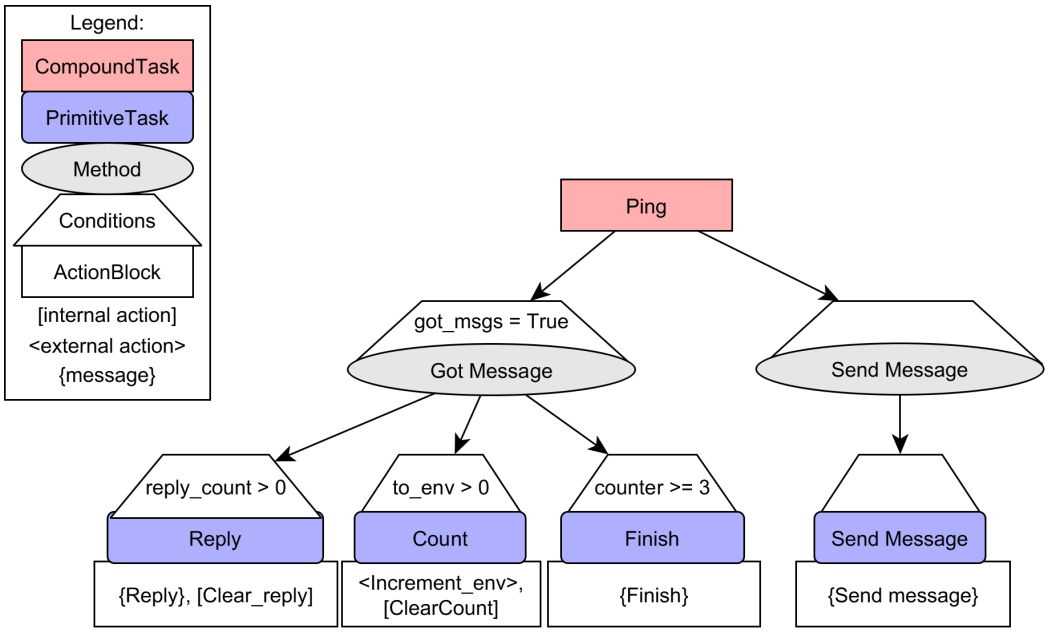

Figure 5.2: HTN for the "ping" test

  - If it contains a "ping" message, the agent replies by a "pong" message.
  - If it contains a "pong" message, it increments the environment's counter and its own counter.

- If no messages were received, the agent sends a "ping" message to another agent that is random chosen from the directory.
- Once the agent's counter value is equal to 3 or more, the agent finishes its execution by sending the "finish" request to the controller.
- The "Got Message" method's primitive tasks' action blocks are combined together.

We expect the following results of this simulation:

- The number of agents diminish until there is no more than one agent.
- Inboxes contain normal messages besides states.
- HTN logic correctly specifies the scenario instructions.
- Actions from the "Got Message" do indeed get executed as part of the same action block.

5.1.1.3 Random messaging scenario

This is the most complex of functionality tests. The following list contains the rules of the scenario.

- The simulation contains 100 agents
- Simulation lasts for a total of 100 steps, and is repeated 10 times.
- Agents' behaviors are guided by a trivial HTN with 4 actions (Fig.5.3):

  Decide internal action is used to randomly pick a number of messages to be sent

  Send message action is used to send the said number of messages to randomly chosen agents from the directory

  Write external action is used to send to the environment the number of messages sent and received

  Cleanup internal action is used to reset the said numbers

- In each simulation step each agent sends from 5 to 20 messages (randomly chosen) to a random agent each, except itself.
- Each agent counts the number of messages it received
- The number of messages sent and received are written to the environment that acts as a blackboard.
- A special modification was made to the `agent_step` function that counts the time it takes for each agent to finish its step.
- Environment's poststep writes the collected values to the log file for further analysis

We expect the following results:

- In the environment the total number of messages sent on each step is equal to the total number of messages received on the next.
- Agents' execution duration are approximately the same, without clear outliers

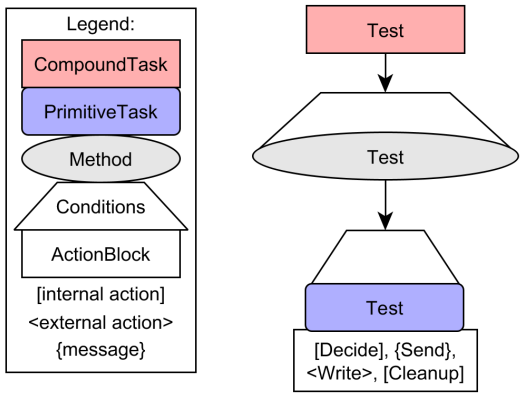

Figure 5.3: Random messaging test HTN

5.1.2 Performance tests

The overall time performance of the system is evaluated in accordance with the 4 parameters:

- Number of agents
- Number of messages sent
- Size of messages sent (given as a number of `0` integers in the list attached to the message)
- Number of requested processing units

For this test cases a special simulation was designed:

- Agents' behaviors are guided by a trivial HTN with a single compound task, single method, and single primitive task with 2 actions: send messages and increment step counter belief.
- On each turn each agent sent a specified number of messages containing lists of zeroes of specified length to random agents and incremented its "step" counter belief.
- Upon the reception of a message, each agent incremented its "counter" belief
- For each series of tests one of the parameters was modified while the others were fixed at default values.

- The default values were:
  - Number of agents: 100
  - Number of messages: 10
  - Message size: 0kb
  - Number of processes: 256

For these tests we expect the following results:

- The performance is supposed to be linearly dependent of the number of agents. Additionally, due to the fact, that all the tasks get submitted almost at the same time, we may see convex fragments between the lightly peaked points that roughly correspond to the multiples of the number of processes. This is due to the fact, that once we schedule the tasks for all the processes, the remainder will have to wait until these tasks finish, which they should do almost at the same time.
- Performance is expected to be sublinearly dependent of the number of messages and the size of the messages
- With the increase of the number of processes the performance should increase until we reach the number of processes equal to the number of agents (tasks). After that the changes are expected to be minimal.

5.1.3 River-basin simulation

One of the current real-world applications of the system is the simulation of the wastewater production and processing in the context of a river basin. This version of the scenario is abstract, but the goal is to apply it to simulate the interactions of factories and treatment plants with the Besòs river in Catalonia.

In our model we represent a river as an ordered list of divisions which we call sections. Each of these sections is a 1 kilometer long part of the river. We then assume the river works in a stationary state and, therefore, each section contains a water mass that represents the mass of water that is held in that section.

We then assume water masses are characterized as: $W = \langle V, C^r \rangle$, where:

- V represents the volume of water

- C^r represents pollutant concentrations and for this scenario we are only considering the following pollutant indicators: Suspended Solids (SS), Biological Oxygen Demand (BOD), Chemical Oxygen Demand (COD), Total Nitrogen (TN), and Total Phosphorus (TP).

We can then express C^r as a set of the five pollutants mentioned above:

$$C^r = \langle C^1, C^2, C^3, C^4, C^5 \rangle = \langle SS, BOD, COD, TN, TP \rangle$$

V is expressed as m^3 and C^r are usually expressed as g/m^3. [1]
The full scenario we want to simulate is composed of:

- A stream-lined river with 40 sections. We assume that all of them are initialized with fresh water masses (i.e., $W = \langle 5, \langle 0, 0, 0, 0, 0 \rangle \rangle$)
- Two Wastewater Treatment Plants (WWTPs)
- Two towns (households)
- Forty industries, divided into in the following productive sectors:
  - Chemical (8)
  - Slaughterhouse (8)
  - Paper mill (8)
  - Furniture plant (6)
  - Plastics (10)

Households and industries are the main wastewater generators in this scenario, as well as the river's headstream. Wastewater is considered as a water mass with significant pollutant concentrations. In order to maintain the water quality of the river and make sustainable use of it, Wastewater Treatment Plants handle wastewater and treat it so that it can be safely discharged into the river. The system also contains retention tanks, which collect water when it rains and discharges it in a controlled way by laminating the water inflows into the sewer system (see Fig.5.4). Some of these tanks are also used for industrial discharges. In the following sections, each of these elements is described and specific data is provided for the simulation prototype.

[1] Sometimes however we express certain values in liters ($1\ m^3 = 1 \cdot 10^3$ L) and kilograms of pollutant, which requires multiplying the concentration and volume to obtain that magnitude and then transform from grams to kilograms ($1\ Kg = 1 \cdot 10^3$ g).

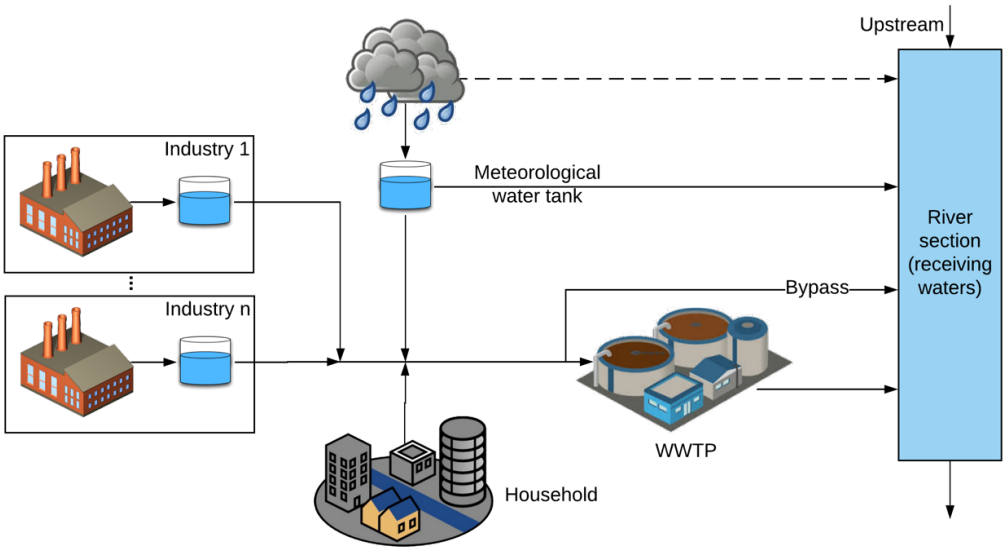

Figure 5.4: Overview of the river

5.1.3.1 Households

Households act as wastewater generators. In the most realistic scenario they automatically discharge a certain volume of wastewater mass at each step; this volume and the pollutants depend on the time of the day. For simplicity, we assume that volume and pollutant concentration does not vary along the day and households always discharge the same water mass. However, depending on the population of each town, the characteristics of such wastewater mass will change accordingly.

For our scenario we consider two towns:

- Coal Hill
  - Population: 3584
  - Location: near section 3
  - Wastewater: $\langle 16.875, \langle 100, 130, 260, 32, 5 \rangle \rangle$
- Torchwood
  - Population 27645
  - Location: near section 36
  - Wastewater: $\langle 130.162, \langle 200, 200, 400, 36, 7 \rangle \rangle$

Households discharge to the nearest WWTP (in terms of number of sections from the household location to the WWTP). This discharge is done at every step. Though the parameters of each discharge is constant for the current scenarios, we force the household agents to report the parameters of the discharge to the WWTP.

The HTN of the current version of Households is shown in Fig.5.5. Though the HTN in this case is trivial (and it can be replaced by the default block), we intentionally keep it as HTN for possible future scenarios.

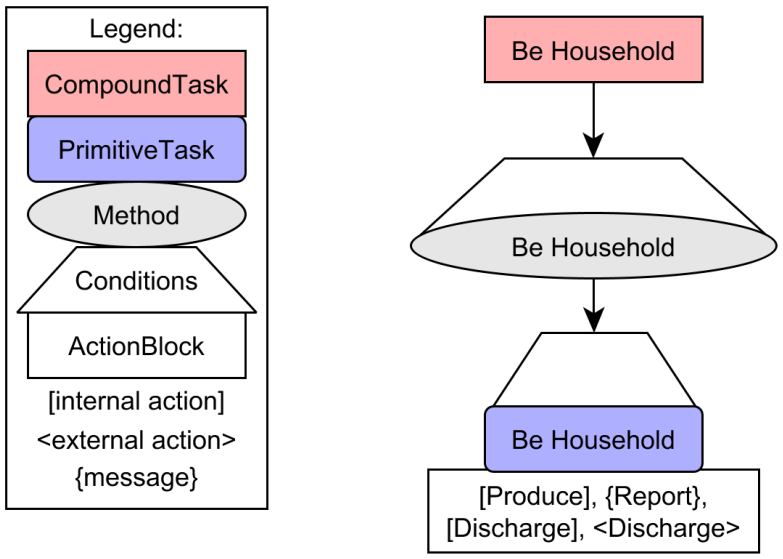

Figure 5.5: HTN for Household agents

5.1.3.2 Industries

Industries manufacture products, generating wastewater as byproduct. We assume that these products are automatically sold and, therefore, they are instantly transformed into profit for the industries. Also, we do not consider actual manufacturing processes and resources (like clear water) demand. Industries have storage tanks to store the wastewater but these are limited and the wastewater has to be discharged into the sewer system (and hence to a WWTP). However, they cannot discharge whenever they want, as explicit permission from a WWTP is required first. In order to get such permission, industries have to pay to the WWTP the cost of treating their wastewater, so it can be discharged into the river safely (i.e. returning the water into the water cycle in a sustainable way).

An industry belongs to one of the following sectors:

- Slaughterhouse
  - Wastewater (per Tn produced): $\langle 1.6 - 6, \langle 422, 450, 986, 59, 22 \rangle \rangle$
  - Location: Randomly chosen between 1–40
  - Storage capacity: 100
- Paper mill
  - Wastewater (per Tn produced): $\langle 100 - 250, \langle 300, 275, 580, 25, 5 \rangle \rangle$
  - Location: Randomly chosen between 1–40
  - Storage capacity: 100
- Plastics
  - Wastewater (per Tn produced): $\langle 2.04, \langle 350, 277, 1200, 50, 6 \rangle \rangle$
  - Location: Randomly chosen between 1–40
  - Storage capacity: 100

For simplicity, we consider that each industry type produces the wastewater following the same parameters. Each sector has a specific pollutant concentration and a given volume (or a range of volumes). Also, while concentrations are fixed, the tons produced by industries are not. This will respectively influence the volume of the produced wastewater (as well as the profits of the industries). Volume depends on the number of tons produced.

The geo-location of an industry is used to determine to which exact WWTP they are connected through the sewer system (we assume they are connected always to the closest one).

Industries try to produce as much as they can at each tick but they have to see to it that:

1. They are not producing more than their maximum production capacity.
2. They are not going to produce more tons if there is a risk of overfilling the current available space in their storage tanks with byproducts.

As it was noted at the start of this section, industries are obliged to request permissions from WWTPs to discharge their byproducts to the sewer system. For each request, the corresponding WWTP will reply with either a refusal

or a permission. In the latter case the response will also contain the cost the industry has to pay to get its wastewater processed. If the industry accepts, the money is detracted automatically from its account, the industry informs the WWTP about the discharge and proceeds with it (thus, also modifying its storage tanks fill status).

It is not necessary for the industry to discharge everything stored.

The current version of an industry's HTN is shown in Fig.5.6

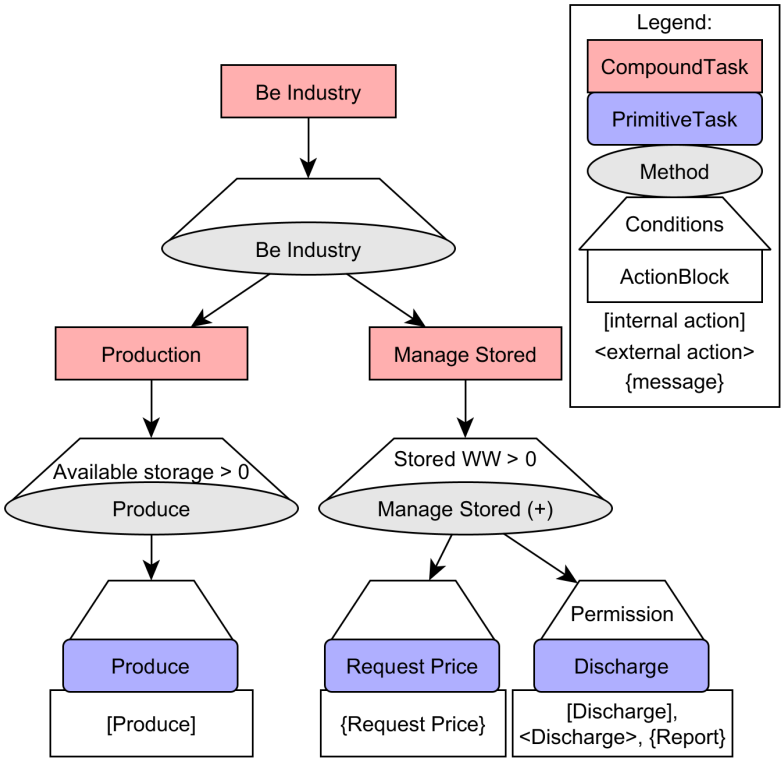

Figure 5.6: HTN for Industry agents

5.1.3.3 Wastewater Treatment Plants (WWTP)

These plants receive wastewater inflows from the sewer system, they treat it, and then they discharge it into the river. This reduces the pollutants concentration in the river and possibly prevents potentially harmful consequences, such as eutrophication or massive algae expansion, which would consume oxygen from the water leading to massive deaths of other living beings in the river [?].

For our scenario we consider two WWTPs:

- WWTP 1
  - Location: 3
  - Storage capacity at entrance/exit: 100/100
  - Treatment capacity: 100
  - Operative threshold: 0.1
  - Pollutant treatment cost (SS/DBO/DQO/TN/TP): (200/150/150/300/300) €/kg
  - Volume treatment cost: 100 €/m^3
- WWTP 2
  - Location: 36
  - Storage capacity at entrance/exit: 100/100
  - Treatment capacity: 100
  - Operative threshold: 0.1
  - Pollutant treatment cost (SS/DBO/DQO/TN/TP): (200/150/150/300/300) €/kg
  - Volume treatment cost: 100 €/m^3

In our model, wastewater arrives from different sources and its masses are merged into the WWTP entrance segment of the sewer system. That tank has a limit; if that limit is reached, all of the surplus wastewater is automatically bypassed to the river. This could happen under different circumstances:

- A sudden amount of wastewater arrives because of precipitations.
- Industrial discharges are higher than expected.
- Both.

The WWTP takes some part or all of the wastewater at the entrance and moves it into the plant for treatment. This treatment takes some time and will remove a predetermined percentage of pollutants. The water will be kept until the process ends and, once that happens, the treated water mass will be discharged into the river section where the WWTP is located.

Additionally, the WWTP has to log the average pollutant concentrations at the entrance and exit so that a performance indicator can be computed. Ideally, WWTPs have to keep their performance at a certain level.

Another important thing about WWTPs is that the treatment is performed with the usage of colonies of bacteria. These colonies require nutrients obtained from the wastewater, therefore the WWTP needs a minimum flow of inflows to remain operative. This requirement is represented by the operative threshold parameter. It denotes the minimal fraction of a WWTP's capacity that has to be filled with wastewater (e.g., treatment capacity 10000 and operative threshold 0.1 would mean that the corresponding WWTP is required to have 1000 m^3 of wastewater stored at treatment at all times).

Treating wastewater has a cost for each kilogram of pollutant extracted from the water. This cost is different for each pollutant and is set as a model parameter.

Also, as mentioned before, treating wastewater takes some time that is expressed in terms of simulation ticks as treatment duration. The process starts with whatever wastewater mass is at the entrance. It then enters and is kept separated from other wastewater masses being treated for the number of ticks equal to the duration of the treatment. After that amount of time has passed, the corresponding wastewater mass has its pollutant concentrations reduced at a given rate. Afterwards, this wastewater mass is discharged into the river.

Besides the treatment itself and the bypassing of decisions, the WWTP has to answer requests for permissions to discharge wastewater. It has to determine if it is feasible to accept a given wastewater discharge and at what cost according to the current load (in terms of volume and pollutant amounts that will be removed).

The WWTP estimates the price of treatment of a wastewater mass (P) as:

$$P = V_i \cdot G \cdot \left(\sum\right)_{r=1}^{5} C_i^r \cdot q^r \cdot g_i^r \tag{5.1}$$

q^r are as follows:

- SS: 50 euros per kg
- DBO: 75 euros per kg.
- DQO: 75 euros per kg.
- TN: 120 euros per kg.
- TP: 150 euros per kg.
- Volume: 100 euros per m^3

It may accept wastewater mass only if $V \geq V_i$, otherwise the request is automatically rejected (V here is the available storage volume in the WWTP) .

At the moment, g_i^r are ignored (set to 1) since we consider all the pollutants equally. However, in more complex scenarios that is usually not a case as higher concentrations cause problems to the bacteria colonies in the WWTP, incurring in higher processing costs.

G is a penalty factor applied by the WWTP that could be a function of the available volume (so that the WWTP increases the cost as the free storage volume is reduced). Note that WWTP also has to try to keep a minimum storage volume available for any unexpected influents.

The HTN for WWTPs is shown in Fig.5.7

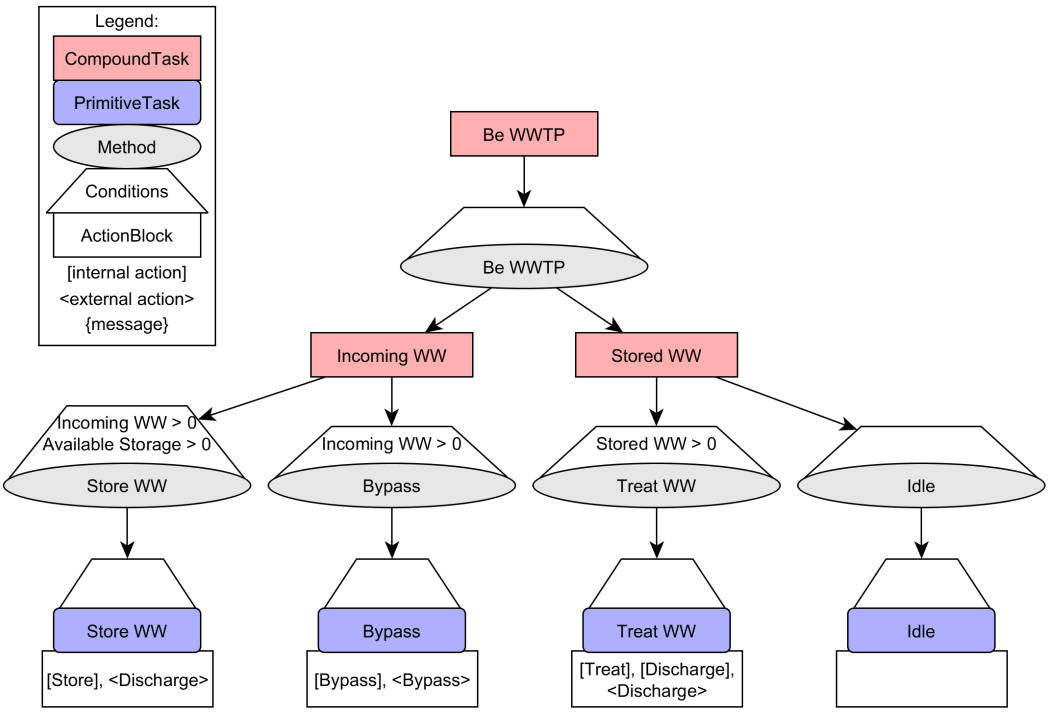

Figure 5.7: HTN for WWTP agents

5.1.4 Environment

The environment is composed of the aforementioned river, the sewer system, and of rains. For the purpose of this scenario and for simplicity, precipit-

ations are collected to meteorological water tank abstractions and sent to the WWTP.

Currently, we consider meteorological water tanks as part of the environment. Therefore, meteorological wastewater is collected and then sent during the following step of the simulation.

As described in section 5.1.3, the river is divided into sections of 1 km each, and we assume that it works in a stationary state. This implies that sections are defined by the watermass they hold and those masses move from one section to the following one at each simulation step. This movement reduces pollutant concentration by a given percentage:

- SS: 20
- DBO: 15
- DQO: 15
- TN: 10
- TP: 5

When different water masses collide, they are merged using the standard chemical formulas for concentration mixing:

$$\frac{c_1 V_1 + c_2 V_2}{V_1 + V_2}$$

where c_1 and c_2 are concentration values, and V_1 and V_2 are the corresponding volumes of the watermasses.

Sewers are represented by parallel "river" segments, leading from the segments within the range of the WWTP to this WWTP. The WWTP then perceives the wastewater at the final entrance segment and either removes it (taking it for treatment), bypasses it, or does a mix of both.

Note that water inside of the sewer system does not clean itself as the water in the river does.

5.1.5 Test scenario

For testing agents' interaction and communication in our platform, we have simplified the model by removing high-detail elements. This eases not only the task of implementation but facilitates the interpretation of the results obtained. This simplification entails not considering the rain in the proposed scenarios as well as ignoring the presence of households. Next subsections provide a thoroughly description of these scenarios

5.1.5.1 One industry and one WWTP

The scheme for these scenario is depicted in Fig.5.8

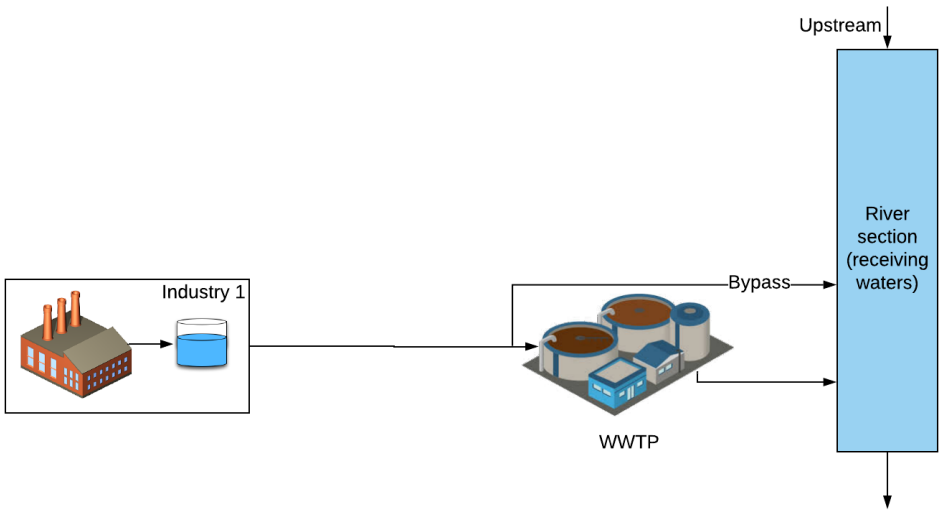

Figure 5.8: Scheme of the test scenario

For the purpose of this scenario the river only has a small amount of sections (4). The industry belongs to the slaughterhouse sector:

- Slaughterhouse
  - Wastewater (per Tn produced): $\langle 1.6 - 6, \langle 422, 450, 986, 59, 22 \rangle \rangle$
  - Location: 1
  - Storage capacity: S_{ind}
  - Maximum production: P_{ind} Tn/hour
  - Euros per Tn produced: 500 €/Tn = B

The WWTP discharges into the first section. Its details are:

- WWTP
  - Location: 1
  - Treatment time: 5 hours (ticks)
  - Storage capacity at entrance/exit: S_{wwtp}

- Treatment capacity: S_{wwtp}· Treatment time
- Treatment cost: C €/m^3
- Operative threshold: 0

We also assume that there is no rain.

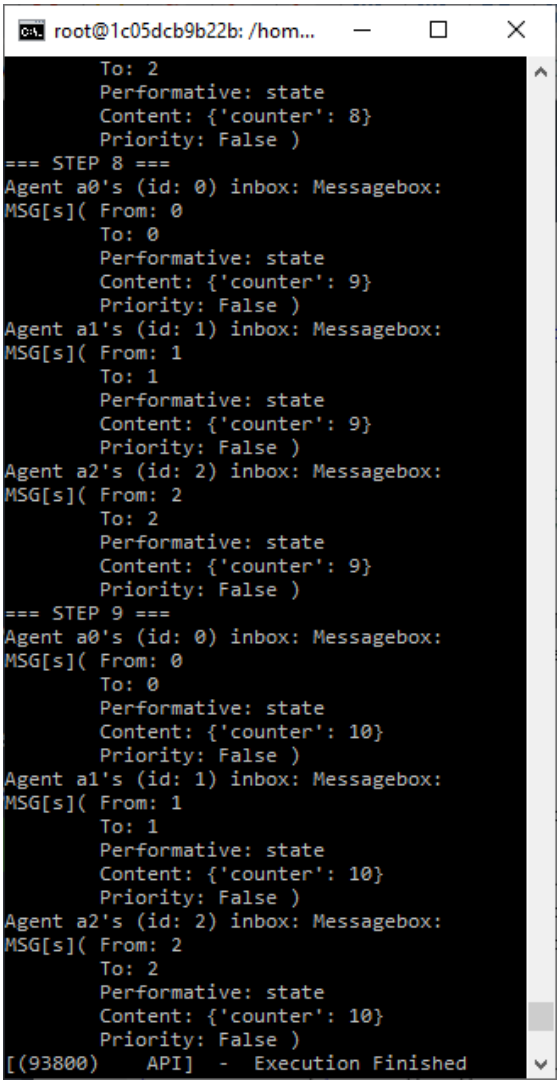

Figure 5.9: Output logs for the incrementation scenario

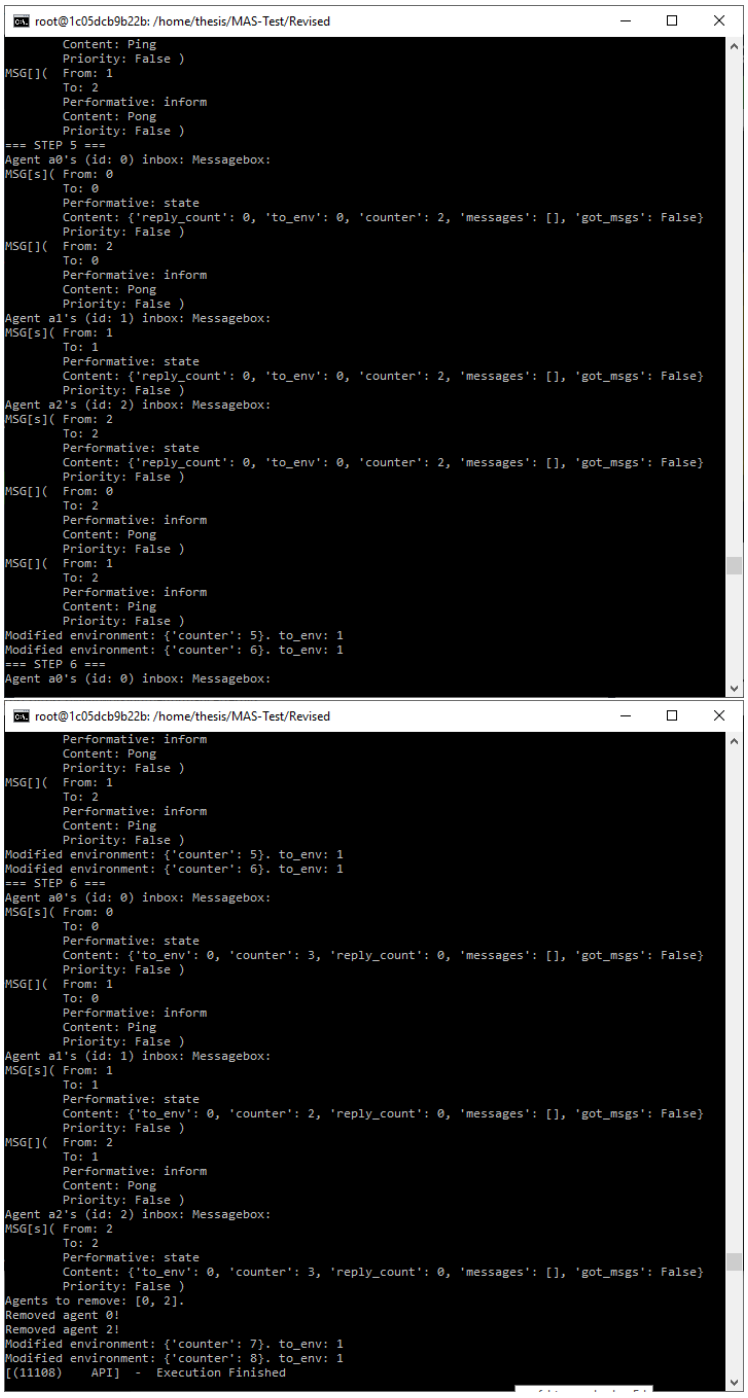

Figure 5.10: Output logs for the ping scenario

5.2 Results

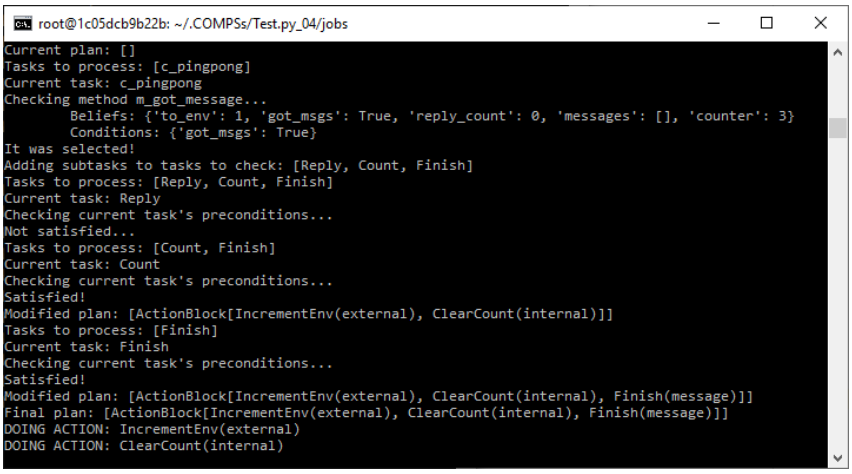

Figure 5.11: Agent step log for the ping scenario

5.2.1 Functionality tests

The results of these tests are shown on Fig.5.9 to 5.12. As it can be seen from the outputs, the MASs in both cases function just as expected:

- In the incrementation test the simulation finished after 10 iterations (numeration is from 0) with the counter value for each agent equal to 10. Moreover, there is no other messages in the agent's inboxes.

- For the ping scenario we see different messages in the agents' inboxes, scenario terminates when two agents is removed (leaving the system with only `Agent 1`), the removed agents have `counter` belief equal to 3. Fig.5.11 shows part of a PyCOMPSs log file corresponding to one step of one agent: there we can see the planning agent's reasoning stage and also that it executes during the same step actions that are originally from different action blocks, proving the fact of their merging.

- For random messaging scenario we have an environment logs and task duration distribution. The former contains lines of numbers in form "`messages_sent messages_received`", the latter is the standard histogram with 100 bins. From the part of the environment log we can see that the number of the messages sent on the previous step is equal to

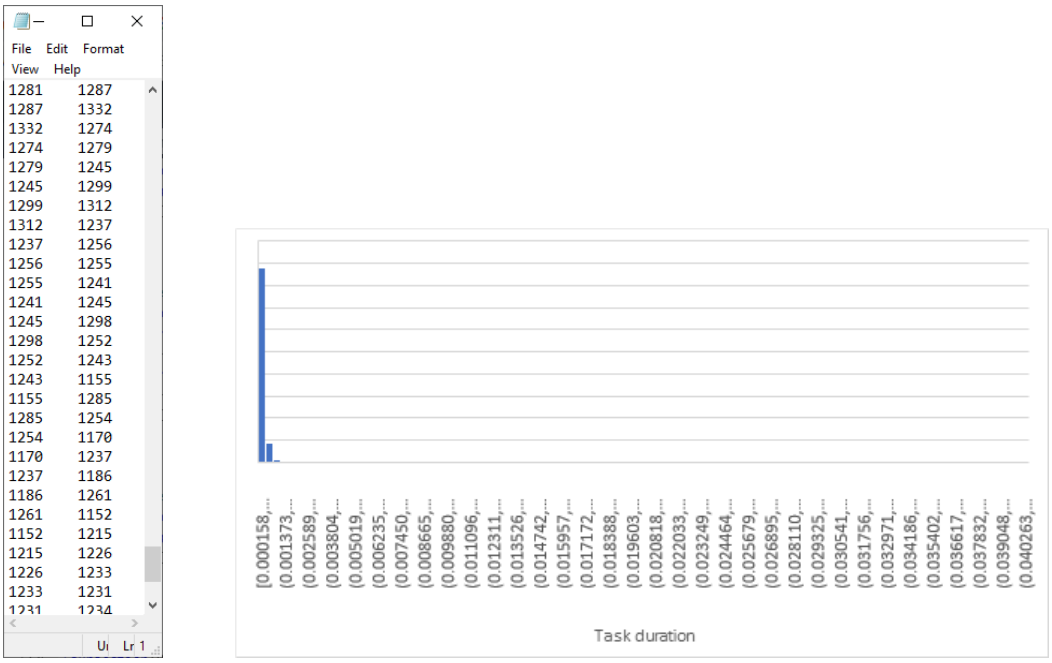

Figure 5.12: Environment values and task duration logs for random messaging scenario

the number of messages received on the next step. Simple verification script confirms the results for the whole length of the log, with expected exception of the edge cases (when there are no previous step and the next step). Histogram and (the range of values) show that all the tasks are executed almost instanteneously and fast, falling within the range of tens of milliseconds.

5.2.2 Performance tests

All of these experiments, with two exceptions, were performed on BSC NordIII cluster using in the default setting 256 processors cores per test. Each core is Intel SandyBridge-EP E5–2670 at 2.6 GHz. All the results are averages of 5 replications.

The time results are presented as triples of total time the test took, part of that time that was spent on task computations, and part of that time that was spent by the controller (for message forwarding and action execution).

Besides the results on the tested metrics we have obtained additional insights in the possible weaknesses of the system, and confirming the directions for the future research. The main issue we have faced was the disk space us-

age. As COMPSs transforms custom objects into files for transfer, this puts a strain on the data transfer infrastructure and may result in exceeding cluster disk memory quotas. This has limited the scale of the results we were able to achieve (although they still exceed the standard capabilities of the non-HPC BDI platforms). Data persistency via dataClay that we were mentioning in § 2.2 is the solution for the issue and we discuss it as a future research in § 6.3.1.

First, we present the results for message size in Fig.5.13. The message size clearly affect the performance, although the mathematical nature of this is unclear. We assume that the "ragged" nature of the graph may be result of different task over nodes assignment and thus different cost of data transfer.

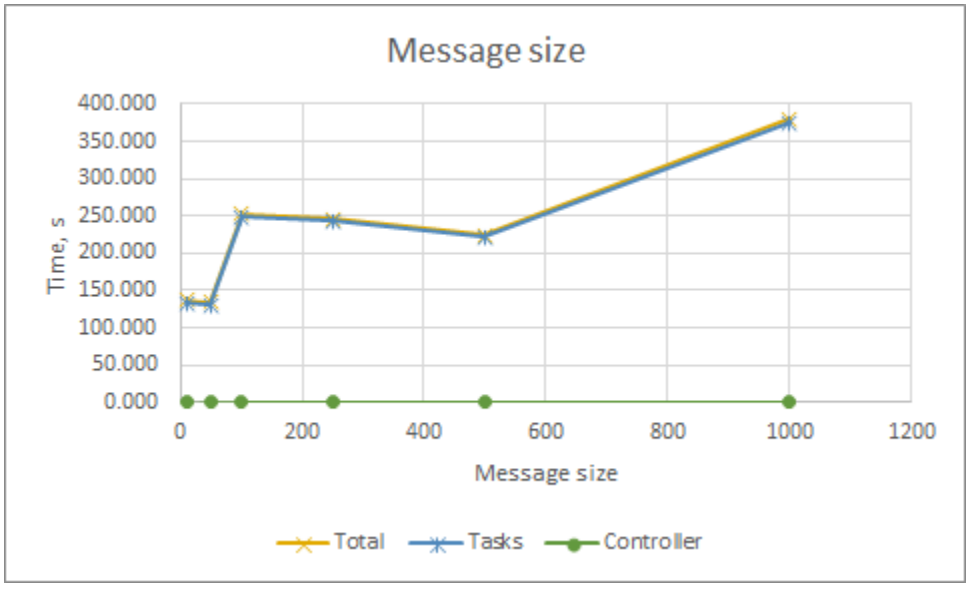

Figure 5.13: Results for message size test

As expected, the controller time is not affected by the size of the messages, as it only deals with the pointers to actual messages.

On the other hand, the results for message count test are clear: Fig. 5.14. We can clearly see the sublinear dependency with increase of message numbers, although we do not have yet enough data to specify their relation beyond that. The sublinear nature of the dependency may be explained by the fact that the messages do not take much disk space, so the most time goes for launching the transfer processes. When all the agents get at least some messages, there is no more need to launch extra processes.

Another observation that was expected is the noticeable increase in con-

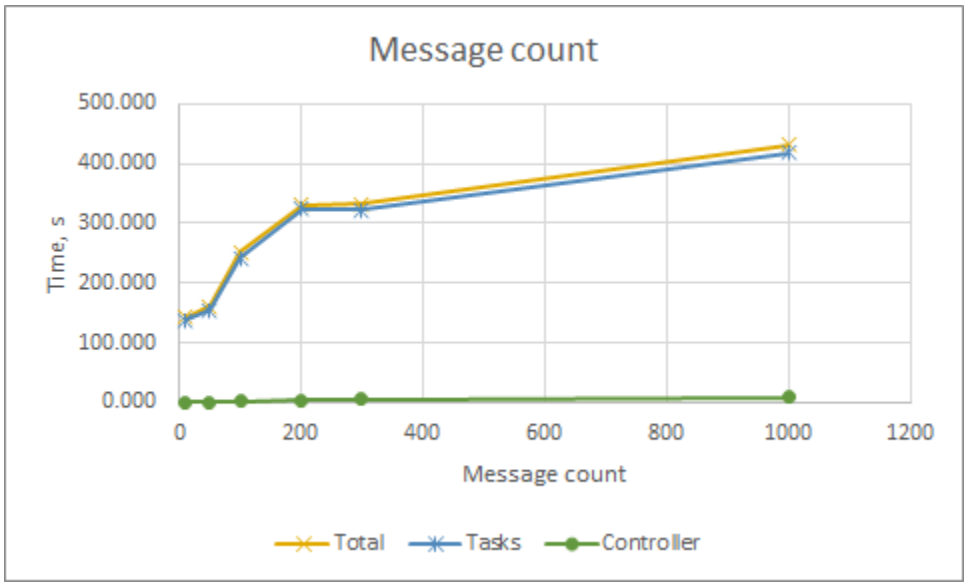

Figure 5.14: Results for message count test

troller time. As the number of messages increase, it have to forward more and more messages, which affects the performance, though still insignificantly in this scenarios.

For the number of agents the behavior of the graph (Fig.5.15) also fits the expectations. The graph increases linearly with increase of the total number of agents (and, correspondingly, the number of PyCOMPSs tasks). We may also notice small convex sections that were also hypothesised although at larger scale.

Also, it is not clear enough from the plot, but the controller time also noticeably increases due to the fact that controller has to process additional messages and handle these new agents.

Probably the most exceptional results can be presented for the experiments on the number of processes, Fig.5.16. For them we have two graphs: for one is the graph for the default tests with specified agent behavior, the other one is for the slightly modified version of it, where on each step we have added a 1 second delay. Both graphs start at number of processes equal to 1, and we can see that in the first case the sequential execution outperforms the distributed one. The reason is that the distribution of tasks and infrastructure interactions take some time. As we have explained in §3.1 due to that COMPSs is not suitable for fin-grained tasks. But as soon as we imitate the harder tasks that take more time to execute, even just one second longer, the difference gets phenomenal.

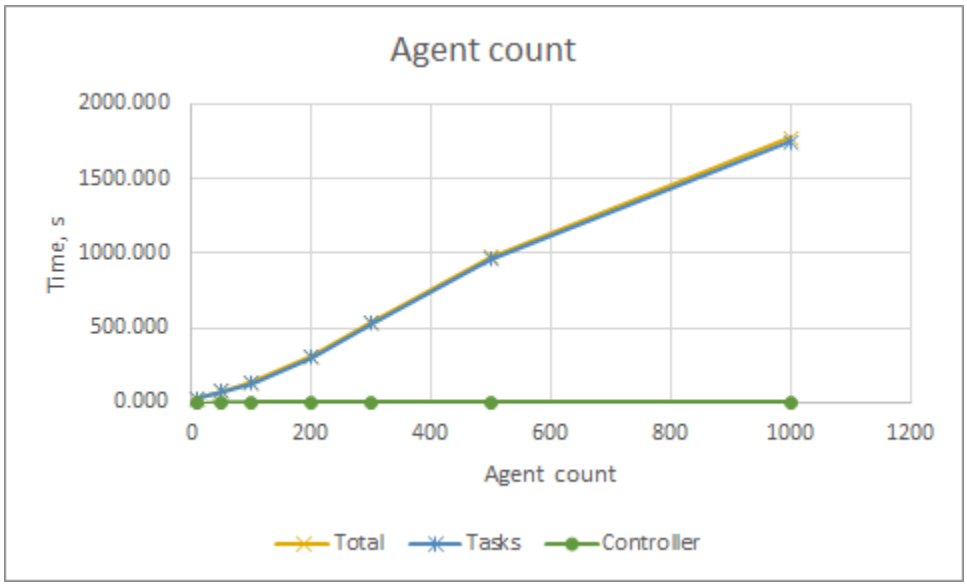

Figure 5.15: Results for agent count test

Also notice that after the number of processes exceed the number of tasks, the changes in the computation time are almost non-existent, as COMPSs have enough resources to get all the tasks distributed with minimal delay.

5.2.3 River basin simulation

River basin have been successfully implemented and tested. The code for the implementation is presented in the Appendix A. Fig.5.17 shows the inner representation of the river that helps to facilitate easy flow of water from industries and households via sewers to WWTPs and from them and headstreams to the end of the river. As we can see, for the case of 2 WWTPs there are essentially 3 river objects: one for the river itself, and one per sewer system connecting to each of the WWTPs.

The HTNs were implemented as shown on the Fig. 5.5, 5.6, 5.7, with their `processing` and `perceive` functions following the scenarios guidelines:

- Industries do not perceive the environment and can get the accept-proposal and reject-proposal messages from WWTPs. In case of the first message, it discharges the specified volume of water to the sewers (or at least as much as it can afford). In case the the second one, it marks the stored water in question as not under the question, forcing to propose its discharge once again.

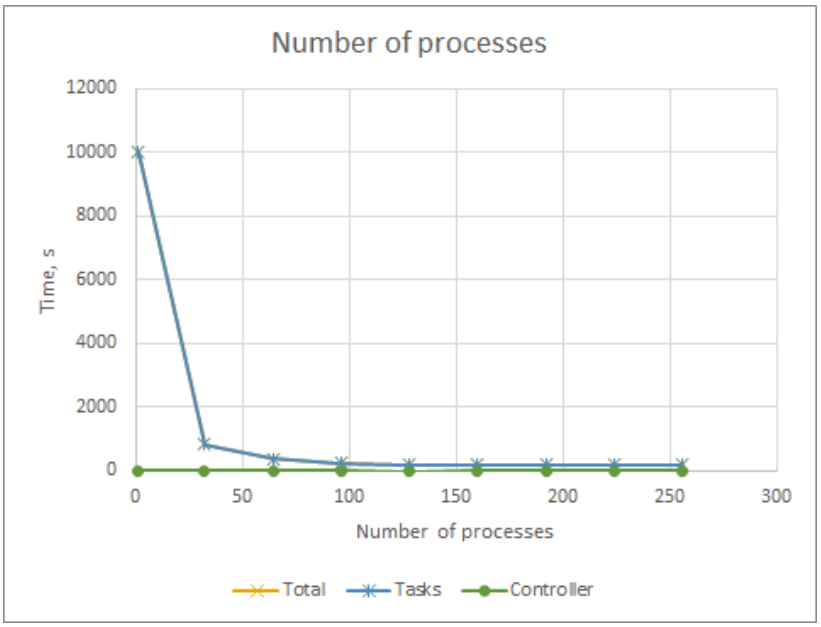

Figure 5.16: Results for number of processes tests. The left graph covers the case of the original agent behaviours, the right one is ones with introduced 1 second delays.

- WWTPs perceive their entrance section of the sewers (in order to later reason about what to do with this water: bypass or store). They can receive inform messages from industries and households (which will modify their expectations of incoming wastewater), and can get the proposals from industries. In case of the latter they either refuse if they cannot accept that much wastewater, or calculate the price according to the formula 5.1.

An example of the state of the river in textual format is shown in Fig. 5.18.

As the goal of this scenario was to test the expressiveness of the system, we have successfully achieved it. Nevertheless, the implementation of the river basin scenario in its most complex form stays as one of the main future applications of the system.

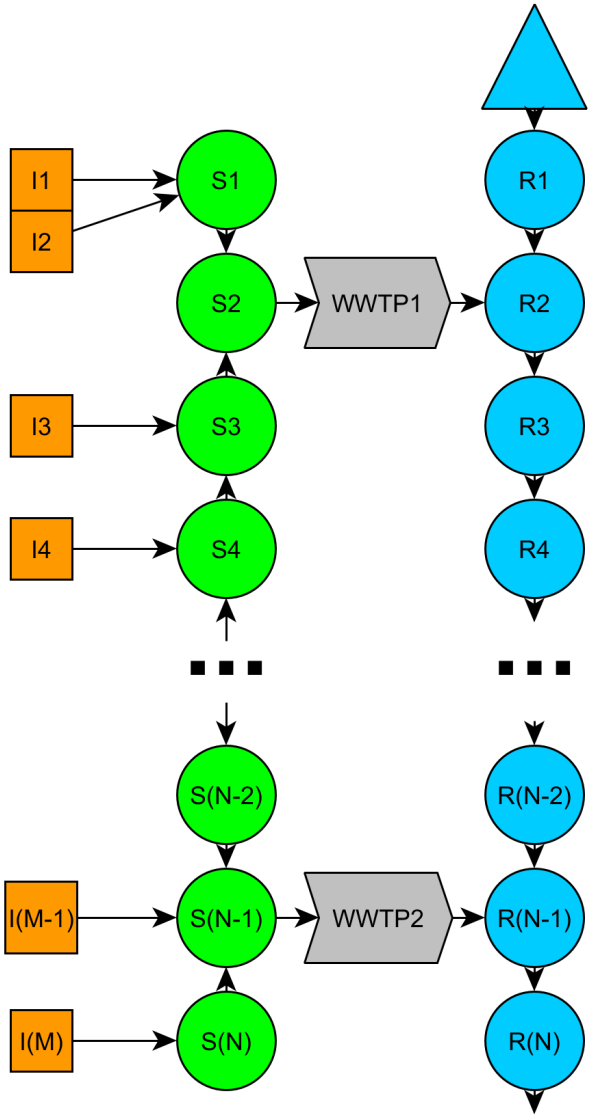

Figure 5.17: Implemented representation of the river

```
45
River
    Section 1: (Volume: 105.0, SS: 200.952380952, BOD: 227.678571429, COD: 498.869047619, TN: 31.6071428571, TP: 12.4404761905)
    Section 2: (Volume: 105.0, SS: 64.3047619048, BOD: 77.4107142857, COD: 169.61547619, TN: 11.3785714286, TP: 4.72738095238)
    Section 3: (Volume: 5.0, SS: 0.0, BOD: 0.0, COD: 0.0, TN: 0.0, TP: 0.0)
    Section 4: (Volume: 105.0, SS: 102.887619048, BOD: 139.823102679, COD: 306.367953869, TN: 23.0416071429, TP: 10.6661532738)
    Section 101: Effluent

Sewers
    Section 401: Effluent
    Section 301: (Volume: 50.0, SS: 422.0, BOD: 450.0, COD: 986.0, TN: 59.0, TP: 22.0)
46
River
    Section 1: (Volume: 5.0, SS: 0.0, BOD: 0.0, COD: 0.0, TN: 0.0, TP: 0.0)
    Section 2: (Volume: 105.0, SS: 160.761904762, BOD: 193.526785714, COD: 424.038690476, TN: 28.4464285714, TP: 11.818452381)
    Section 3: (Volume: 105.0, SS: 51.4438095238, BOD: 65.7991071429, COD: 144.173154762, TN: 10.2407142857, TP: 4.49101190476)
    Section 4: (Volume: 5.0, SS: 0.0, BOD: 0.0, COD: 0.0, TN: 0.0, TP: 0.0)
    Section 101: Effluent

Sewers
    Section 401: Effluent
    Section 301: (Volume: 0.0, SS: 0, BOD: 0, COD: 0, TN: 0, TP: 0)
47
River
    Section 1: (Volume: 105.0, SS: 80.380952381, BOD: 91.0714285714, COD: 199.547619048, TN: 12.6428571429, TP: 4.97619047619)
    Section 2: (Volume: 5.0, SS: 0.0, BOD: 0.0, COD: 0.0, TN: 0.0, TP: 0.0)
    Section 3: (Volume: 105.0, SS: 128.60952381, BOD: 164.497767857, COD: 360.432886905, TN: 25.6017857143, TP: 11.2275297619)
    Section 4: (Volume: 105.0, SS: 41.155047619, BOD: 55.9292410714, COD: 122.547181548, TN: 9.21664285714, TP: 4.26646130952)
    Section 101: Effluent

Sewers
    Section 401: Effluent
    Section 301: (Volume: 150.0, SS: 422.0, BOD: 450.0, COD: 986.0, TN: 59.0, TP: 22.0)
48
River
    Section 1: (Volume: 105.0, SS: 200.952380952, BOD: 227.678571429, COD: 498.869047619, TN: 31.6071428571, TP: 12.4404761905)
    Section 2: (Volume: 105.0, SS: 64.3047619048, BOD: 77.4107142857, COD: 169.61547619, TN: 11.3785714286, TP: 4.72738095238)
    Section 3: (Volume: 5.0, SS: 0.0, BOD: 0.0, COD: 0.0, TN: 0.0, TP: 0.0)
    Section 4: (Volume: 105.0, SS: 102.887619048, BOD: 139.823102679, COD: 306.367953869, TN: 23.0416071429, TP: 10.6661532738)
    Section 101: Effluent
```

Figure 5.18: State of the river during the runtime

5.3 Chapter summary

In this chapter we have focused on showcasing the performance of the developed system. We have proposed experiments to provide performance evaluation of the system, as well as the ones to prove that it works as expected. The experiments helped us to identify possible bottlenecks of the current implementation (data transfer) and, in general terms, showed that the reasoning cycle and HTN planner run correctly.

In the next and final chapter we will sum up all the work done, go over the initial list of objectives, and provide an extensive discussion of further development of the system.

Chapter 6

Conclusions

In this document we present a model and an implementation of an Agent-Based micro-simulation framework that can be effectively executed on an HPC environment and can support complex, BDI-inspired reasoning in its agents. This concluding chapter provides an analysis of the work done and some points to future lines of research. First of all, in section 6.1 we revisit the Master Thesis objectives to show how they have been fulfilled; then, in section 6.2 we discuss how our model and our system contributes to the development of the field of Agent-based micro-simulation; finally, in section 6.3 we discuss how the system could be improved even further and what are the upcoming features.

6.1 Revisiting objectives

The original set of objectives that were set for this work was:

1. To study the COMPSs framework for distributed computing and its use for implementing a micro-simulations platforms under it.
2. To study the SPADE agent platform to assess its applicability in the HPC domain.
3. To develop a general multi-agent-based simulation model for the proposed framework.
4. To introduce a reasoning cycle to it.
5. To implement an agent platform based on this model.
6. To test the system in terms of performance and develop test simulations.

In this section we will summarise the advances on every objective from this list.

6.1.1 COMPSs study

The COMPSs framework has been thoroughly studied. Its approach to distribution of tasks in sequential programs was laid as a foundation for the proposed agent-based HPC simulation framework. Thus, we have also shown that it is possible to use it for micro-simulations on clusters.

6.1.2 SPADE

We have performed an in-depth analysis of SPADE documentation and the current and the previous version of the code. We have also prepared a report on the use of external libraries by SPADE. The found inter-dependencies has proven to be incompatible with the cluster-based HPC applications in general and with COMPSs framework in particular.

To cover for this failed objective we switched to our own simple agent package for testing purposes. However, later on we have adopted and developed the aforementioned package as the main part of the implementation of the framework.

6.1.3 Model for the framework

We have designed a model for the agent-based HPC simulation framework based on COMPSs properties. Furthermore we have provided formal definitions that cover all the main components of the framework int he form of transition rules that describe the agents' deliberation cycles and controller's functions. This formalisation of the framework is covered in depth in § 3.3.

6.1.4 Reasoning cycle

We propose a BDI-inspired agent deliberation cycle and integrate it into COMPSs elements. The reasoning cycle is also described as part of the framework's formalization in § 3.3. In addition, § 3.4 and § 4.2 cover the basic instantiation of the reasoning function we use in the implementation of our model: the HTN planner. We have studied some HTN models and have chosen one that was a great fit for out problem setting.

6.1.5 Implementation

The proposed framework was implemented in Python and is based on the PyCOMPSs package. It deals with various covered aspects of agents' behaviours, add some additional experimental features and provides the user implicit access to PyCOMPSs. Though there is a considerable number of possible points of improvements, the system already provides all the necessary functionality. The agent behaviours are supposed to be coded by the user in Python using the classes provided by the framework and, possibly, (but with some consideration) custom or third party ones.

6.1.6 Testing

A set of experiments was designed to prove the correctness, effectiveness, and efficiency of the implementation. We have shown that the system can improve the computation times of the model depending on the requested number of nodes and demonstrating the scaling of the system of regards to such parameters as messaging intensity, message sizes, and the number of agents. We have also uncovered some limitations in form of disk memory which will be covered in the future work by integration with dataClay.

Finally, we have introduced a real-world scenario: simulation of wastewater production and treatment on the river scale.

6.2 Contributions to Artificial Intelligence

This work contributes to the field of artificial intelligence in a number of ways.

First of all, this work presents the first formalised model of a framework for BDI-like agent micro-simulations on HPC. This kind of simulation is centered on modelling complex processes by modelling the individual units or actors participating in it. As the result, we propose a framework that allows to efficiently study complex compound problems in bottom-up manner.

Additionally, we present the first implementation of such framework on top of COMPSs framework. Thus, we show that they can actually work together and that it is possible to have a concrete version of such models simulated on clusters.

Finally, we have made a first implementation of a simple social HTN planner that can be parallelised by COMPSs. By collectively selecting goals and following plans on them we can achieve shared goals, and the whole process can be done part by part on different computation nodes.

6.3 Future lines of research

There are several possible areas of improvement of the system, differing in scale.

6.3.1 Data persistence

In the current version of the system the controller is a bottleneck. It handles the messages, the environment, the tasks, and the main loop. One way to reduce the system's dependency of it is to use a data persistence framework to reduce the involvement of the controller in data transfers. Additionally, this would increase the fault-tolerance of the system.

During the timeline of this Master Thesis we already explored the integration of a data persistence framework available for COMPSs: dataClay [47]. Integration work already started, but is not finished due to some internal server setup issues which are out of scope of this work. The integration work is planned to continue in the next months, with the collaboration of both the COMPSs and dataClay teams at BSC.

A lot of parts of the Agent-based HPC simulation system may benefit from it. For instance, this would allow to exchange messages without a proxy-controller, agents would just need to push outgoing messages directly to the receiver agent's persistent inbox. Agent's states would also benefit from the persistence mechanism. Instead of moving states through the controller, they may be kept on the nodes and synchronised when the agent's node change. Also directory and the environment may be broadcasted or made public by making them persistent.

6.3.2 Multi-role management

The current version of the formal conceptual model for our agent-based HPC simulation framework and its implementation support the change of the role during the simulation. However, a good addition to that would be to allow agents to have multiple active roles that these agents can reason about in parallel, without the need to drop completely one role to be able to reason w.r.t. another one. Moreover, we would like to create some mechanisms to activate and deactivate these roles, switch between them and compose behaviours coming from the enactment of several (compatible) roles at the same time.

6.3.3 Introduction of social norms in simulations

Social norms usage is a rather promising practice for reasoning agents in social setups. Norms can be used as social rules denoting what should and what should not be done, and what are the expected consequences for various actions. In complex environment with a lot of actions to choose from, consulting about the effect of agent's action in terms of norm enforcement may significantly reduce the size of the search space for further planning.

During this Master Thesis timeline we have stated discussions with Dr. Julian Padget to explore the integration of the InstAL framework on top of ours. The InstAL system [19][52] provides norms analysis and interpretation as a service. Due to this it may be feasible to incorporate it in the proposed system and provide another layer of functionality for the users.

6.3.4 Full ACL support

As of now, our framework supports only parts of FIPA ACL (Agent Communication Language). For instance, the messages are independent of each other and though one may compose a message as a response to another message, no communication act identifiers are built-in to allow the management of conversation chains and protocols.

Moreover, it would also be useful to develop a pattern-based multiple message processors selection mechanism.

Introducing full ACL compliance and support will allow the use of the mentioned features and standardise the messaging patterns in the simulation.

6.3.5 Support standard PDDL and HTN plan files

As presently the implementation of the framework uses custom programmatically defined HTNs. The system would greatly benefit from using the some standard formats for saving some of its parts. This will increase the compatibility of our system with the others, and will allow to upload and use as-is or with minimal additions custom or third-party HTNs for HPC microsimulations.

Bibliography

[1] https://www.bsc.es/research-and-development/software-and-apps/software-list/comp-superscalar/documentation. Accessed on 15-04-2019. 30

[2] Blue brain project. https://www.epfl.ch/research/domains/bluebrain/. Accessed on 07-04-2019. 1

[3] The foundation for intelligent physical agents. http://www.fipa.org/. Accessed: 2019-01-23. 8, 53

[4] Human brain project. https://www.humanbrainproject.eu/en/. Accessed on 07-04-2019. 1

[5] Netlogo. https://ccl.northwestern.edu/netlogo/. Accessed on 15-04-2019. 15

[6] Basic features of the grid component model (assessed). CoreGRID Deliverable D.PM.04, 2007. 23

[7] Sameera Abar, Georgios K. Theodoropoulos, Pierre Lemarinier, and Gregory M.P. O'Hare. Agent based modelling and simulation tools: A review of the state-of-art software. *Computer Science Review*, 24:13 – 33, 2017. 15

[8] R.J. Allan. Survey of agent based modelling and simulation tools. 01 2009. 15

[9] Stefania Bandini, Sara Manzoni, and Giuseppe Vizzari. Agent based modeling and simulation: An informatics perspective. *Journal of Artificial Societies and Social Simulation*, 12(4):4, 2009. 7

[10] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. Jade – a fipa-compliant agent framework. *Telecom Italia Internal Technical Report.*

Part of This Report Has Been Also Published in Proceedings of PAAM'99, 01 1999. 8, 10

[11] Rafael H. Bordini Jomi Fred Hübner Michael Wooldridge Rafael H. Bordini Jomi Fred Hübner Michael Wooldridge Rafael H. Bordini Jomi Fred Hübner Michael Wooldridge Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming Multi[U+2010]Agent Systems in AgentSpeak using Jason*. John Wiley & Sons, Ltd, October 2007. 2, 12

[12] A. Bracciali, N. Demetriou, U. Endriss, A. Kakas, W. Lu, P. Mancarella, F. Sadri, K. Stathis, G. Terreni, and F. Toni. The kgp model of agency for global computing: Computational model and prototype implementation. In Corrado Priami and Paola Quaglia, editors, *Global Computing*, pages 340–367, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. 13

[13] Lars Braubach, Winfried Lamersdorf, and Alexander Pokahr. Jadex: Implementing a bdi-infrastructure for jade agents. *EXP In Search of Innovation (Special Issue on JADE)*, 3, 12 2003. 8, 10, 20

[14] Michael Brenner and Bernhard Nebel. Continual planning and acting in dynamic multiagent environments. In *Proceedings of the 2006 International Symposium on Practical Cognitive Agents and Robots*, PCAR '06, pages 15–26, New York, NY, USA, 2006. ACM. 21

[15] Javier Bueno, Judit Planas, Alejandro Duran, Rosa M. Badia, Xavier Martorell, Eduard Ayguade, and Jesús Labarta. Productive programming of gpu clusters with ompss. pages 557–568, 05 2012. 27

[16] Thomas C. Schelling. *Micromotives and Macrobehavior*. 01 2006. 1

[17] Rafael Cardoso. *A Decentralised Online Multi-Agent Planning Framework for Multi-Agent Systems*. PhD thesis, 03 2018. 20

[18] Jesús Cerquides, Gauthier Picard, and Juan A. Rodríguez-Aguilar. Designing a marketplace for the trading and distribution of energy in the smart grid. In *AAMAS '15: Proceedings of the 14th international conference on autonomous agents and multiagent systems*, Istanbul, Turkey, 04/05/2015 2015. 2

[19] Owen Cliffe, Marina De Vos, and Julian Padget. Answer set programming for representing and reasoning about virtual institutions. In Katsumi Inoue, Ken Satoh, and Francesca Toni, editors, *CLIMA VII*, volume 4371 of *Lecture Notes in Computer Science*, pages 60–79. Springer, 2006. 93

[20] S. Coakley, M. Gheorghe, M. Holcombe, S. Chin, D. Worth, and C. Greenough. Exploitation of high performance computing in the flame agent-based simulation framework. In *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*, pages 538–545, June 2012. 14, 19

[21] Vitor N. Coelho, Miri Weiss Cohen, Igor M. Coelho, Nian Liu, and Frederico Gadelha Guimarães. Multi-agent systems applied for energy systems integration: State-of-the-art applications and trends in microgrids. *Applied Energy*, 187:820 – 832, 2017. 2

[22] Nicholson Collier and Michael North. Parallel agent-based simulation with repast for high performance computing. *SIMULATION*, 89:1215–1235, 10 2012. 14

[23] Nicholson Collier and Michael North. Repast hpc: A platform for large-scale agent-based modeling. *Large-Scale Computing*, pages 81–109, 04 2012. 14, 19

[24] Gennaro Cordasco, Rosario De Chiara, Ada Mancuso, Dario Mazzeo, Vittorio Scarano, and Carmine Spagnuolo. A framework for distributing agent-based simulations. In Michael Alexander, Pasqua D'Ambra, Adam Belloum, George Bosilca, Mario Cannataro, Marco Danelutto, Beniamino Di Martino, Michael Gerndt, Emmanuel Jeannot, Raymond Namyst, Jean Roman, Stephen L. Scott, Jesper Larsson Traff, Geoffroy Vallée, and Josef Weidendorfer, editors, *Euro-Par 2011: Parallel Processing Workshops*, pages 460–470, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. 14, 19

[25] Mehdi Dastani. 2apl: A practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, Jun 2008. 2, 11

[26] Mehdi Dastani, M. Birna van Riemsdijk, and John-Jules Ch. Meyer. Goal types in agent programming. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '06, pages 1285–1287, New York, NY, USA, 2006. ACM. 38

[27] Giuseppe de Giacomo, Yves Lespérance, and Hector J. Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artif. Intell.*, 121(1-2):109–169, August 2000. 13

[28] Giuseppe De Giacomo, Yves Lespérance, Hector J. Levesque, and Sebastian Sardina. *IndiGolog: A High-Level Programming Language for Embedded Reasoning Agents*, pages 31–72. 05 2009. 13, 20

[29] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21:173–193, 06 2011. 27

[30] Michael E. Bratman. Intention, plans and practical reason. *Bibliovault OAI Repository, the University of Chicago Press*, 100, 01 1987. 2, 8

[31] Kutluhan Erol, James Hendler, and Dana Nau. Htn planning: Complexity and expressivity. *Proceedings of the National Conference on Artificial Intelligence*, 2, 05 1994. 11, 20, 36

[32] Kutluhan Erol, James Hendler, and Dana Nau. Umcp: A sound and complete procedure for hierarchical task-network planning. *Proceedings of the International Conference on AI Planning Systems*, 04 2003. 11, 20, 36

[33] Miguel Escriva, Javier Palanca, Gustavo Aranda, Ana Garcia, Vicente Julian, and Vicent Botti. A jabber-based multi-agent system platform. pages 1282–1284, New York, NY, USA, 2006. ACM. 2, 4, 8, 11

[34] Richard Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 12 1971. 12, 20

[35] Michael Fisher. A survey of concurrent metatem - the language and its applications. In *Proceedings of the First International Conference on Temporal Logic*, ICTL '94, pages 480–505, Berlin, Heidelberg, 1994. Springer-Verlag. 13

[36] Michael Fisher. Metatem: The story so far. In Rafael H. Bordini, Mehdi M. Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, editors, *Programming Multi-Agent Systems*, pages 3–22, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. 13

[37] Pezhman Ghadimi, Farshad Ghassemi Toosi, and Cathal Heavey. A multi-agent systems approach for sustainable supplier selection and order allocation in a partnership supply chain. *European Journal of Operational Research*, 269(1):286 – 301, 2018. 2

[38] Lázló Gulyás, Gábor Szemes, George Kampis, and Walter de Black. A Modeler-friendly API for ABM partitioning. In *ASME 2009*, volume 2, pages 219–226, San Diego, California, USA, 2009. 14

[39] Koen V. Hindriks and Tijmen Roberti. Goal as a planning formalism. In *MATES*, 2009. 2, 8, 12, 20

[40] Troy Humphreys. *Exploring HTN Planners through Example*, chapter Architecture, pages 149–167. CRC Press, September 2013. 20, 37, 54

[41] Antonis C Kakas, Paolo Mancarella, Fariba Sadri, Kostas Stathis, and Francesca Toni. The kgp model of agency. In *The 16th European conference on artificial intelligence*, page 3337, 2004. 13, 20

[42] Kalliopi Kravari and Nick Bassiliades. A survey of agent platforms. *Journal of Artificial Societies and Social Simulation*, 18(1):11, 2015. 10, 15

[43] João Alexandre Leite, José Júlio Alferes, and Luís Moniz Pereira. Minerva - a dynamic logic programming agent architecture. In *Revised Papers from the 8th International Workshop on Intelligent Agents VIII*, ATAL '01, pages 141–157, London, UK, UK, 2002. Springer-Verlag. 13

[44] Francesc Lordan, Enric Tejedor, Jorge Ejarque, Roger Rafanell, Javier Álvarez, Fabrizio Marozzo, Daniele Lezzi, Raül Sirvent, Domenico Talia, and Rosa M. Badia. ServiceSs: An Interoperable Programming Framework for the Cloud. *Journal of Grid Computing*, 12(1):67–91, Mar 2014. 3, 4, 19, 23

[45] Alfred J. Lotka. *Elements of Physical Biology*. Williams and Wilkins Company, 1925. 1

[46] S Luke, Gabriel Balan, L A. Panait, Claudio Cioffi, and S Paus. Mason: a java multi-agent simulation library. 04 2012. 14

[47] Jonathan Martí, Anna Queralt, Daniel Gasull, Alex Barceló, Juan José Costa, and Toni Cortes. Dataclay: A distributed data store for effective inter-player data sharing. *Journal of Systems and Software*, 131:129 – 145, 2017. 19, 92

[48] Antonino Marvuglia, Sameer Rege, Tomás Navarrete Gutiérrez, Laureen Vanni, Didier Stilmant, and Enrico Benetto. A return on experience from

the application of agent-based simulations coupled with life cycle assessment to model agricultural processes. *Journal of Cleaner Production*, 142:1539 – 1551, 2017. 2

[49] Nelson Minar, Roger Burkhart, Chris Langton, and Manor Askenazi. The swarm simulation system: A toolkit for building multi-agent simulations. *Santa Fe Institute Working Paper*, 96-06-042, 07 1996. 15

[50] Cynthia Nikolai and Gregory Madey. Tools of the trade: A survey of various agent based modeling platforms. *Journal of Artificial Societies and Social Simulation*, 12(2):2, 2009. 15

[51] Gregory O'Hare, Brian Duffy, Rem Collier, Colm Rooney, and R P. S. O'Donoghue. Agent factory: Towards social robots. 01 1999. 1

[52] Julian Padget, Emad ElDeen Elakehal, Tingting Li, and Marina De Vos. *InstAL: An Institutional Action Language*, pages 101–124. Springer, 2016. 93

[53] Steven F. Railsback, Steven L. Lytinen, and Stephen K. Jackson. Agent-based simulation platforms: Review and development recommendations. *SIMULATION*, 82(9):609–623, 2006. 15

[54] Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a BDI-architecture. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning*, pages 473–484. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA, 1991. 8

[55] Alban Rousset, Bénédicte Herrmann, Christophe Lang, and Laurent Philippe. A survey on parallel and distributed multi-agent systems for high performance computing simulations. *Computer Science Review*, 22:27 – 46, 2016. 15

[56] Xavier Rubio-Campillo. Pandora: A versatile agent-based modelling platform for social simulation. 01 2014. 15, 19

[57] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995. 7

[58] Fariba Sadri. Using the kgp model of agency to design applications. In Francesca Toni and Paolo Torroni, editors, *Computational Logic in Multi-Agent Systems*, pages 165–185, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. 13

[59] Sebastian Sardina, Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. On the semantics of deliberation in indigolog — from theory to implementation. *Annals of Mathematics and Artificial Intelligence*, 41(2):259–299, Aug 2004. 13

[60] Sebastian Sardina and Lin Padgham. A bdi agent programming language with failure handling, declarative goals, and planning. *Autonomous Agents and Multi-Agent Systems*, 23:18–70, 07 2011. 20, 36

[61] Alexander Serenko and Brian Detlor. Agent toolkits: a general overview of the market and an assessment of instructor satisfaction with utilizing toolkits in the classroom. In *DeGroote School of Business Working Paper Series*, number 455, page 43, July 2002. 2

[62] Joseph Suarez, Yilun Du, Phillip Isola, and Igor Mordatch. Neural MMO: A massively multiagent game environment for training and evaluating intelligent agents. *CoRR*, abs/1903.00784, 2019. 2

[63] Y. Tang, X. Xing, H. R. Karimi, L. Kocarev, and J. Kurths. Tracking control of networked multi-agent systems under new characterizations of impulses and its applications in robotic systems. *IEEE Transactions on Industrial Electronics*, 63(2):1299–1307, Feb 2016. 1

[64] Enric Tejedor and Rosa M. Badia. Comp superscalar: bringing grid superscalar and gcm together. pages 185–193, 06 2008. 23

[65] Enric Tejedor, Yolanda Becerra, Guillem Alomar, Anna Queralt, Rosa M Badia, Jordi Torres, Toni Cortes, and Jesús Labarta. Pycompss: Parallel computational workflows in python. *The International Journal of High Performance Computing Applications*, 31(1):66–82, 2017. 4, 19, 27

[66] Enric Tejedor, Jorge Ejarque, Francesc Lordan, Roger Rafanell, Javier Álvarez Cid-Fuentes, Daniele Lezzi, Raúl Sirvent, and Rosa M. Badia. A cloud-unaware programming model for easy development of composite services. pages 375–382, 11 2011. 23

[67] Georgios Theodoropoulos and Brian Logan. A framework for the distributed simulation of agent-based systems. 07 1999. 15

[68] Seth Tisue and Uri Wilensky. Netlogo: A simple environment for modeling complexity. pages 16–21, 01 2004. 15

[69] Richard Tynan, David Marsh, Donal O'Kane, and Gregory O'Hare. Intelligent agents for wireless sensor networks. pages 1179–1180, 01 2005. 2

[70] Vito Voltera. Fluctuations in the abundance of a species considered mathematically. *Nature*, 118(2972):558–560, October 1926. 1

[71] U. Wilensky and K. Reisman. Connectedscience: Learning biology through constructing and testing computational theories – an embodied modeling approach. *InterJournal of Complex Systems*, pages 1–12, 1998. 1

[72] Michael Winikoff. *Jack™ Intelligent Agents: An Industrial Strength Platform*, pages 175–193. Springer US, Boston, MA, 2005. 12, 20

[73] Michael Wooldridge. *An Introduction to MultiAgent Systems*. Wiley Publishing, 2nd edition, 2009. 1, 7

[74] Jiajian Xiao, Philipp Andelfinger, David Eckhoff, Wentong Cai, and Alois Knoll. A survey on agent-based simulation using hardware accelerators. *ACM Comput. Surv.*, 51(6):131:1–131:35, January 2019. 19

Appendix A

HTN example

Here we present the implementation of the HTNs and the behaviors for the river basin scenario.

Household actions:

```
def i_hh_produce(agent, beliefs):
    beliefs["ww_stored"] += beliefs["ww_production"]
    return beliefs


def msg_hh_report(agent, beliefs, directory):
    return [(beliefs["wwtp"], "inform", {"ww": beliefs["
        ww_stored"], "distance": beliefs["distance"]}, False)]


def e_hh_discharge(environment, ww_stored=Water(), section=-1):
    environment["sewers"][environment["sewer_id"][section]].
        sections[section].water += ww_stored
    return environment


def i_hh_discharge(agent, beliefs):
    log = str(beliefs["ww_stored"])
    beliefs["ww_stored"].volume = 0.0
    return beliefs, False, log
```

Household HTN and beliefs initialization:

```
def hh_init(production, wwtp, section, distance):
    ab_be_hh = ActionBlock()
    ab_be_hh.add_internal_action(i_hh_produce, name="[Produce]"
        )
    ab_be_hh.add_message(msg_hh_report, name="{Report}")
```

```
5     ab_be_hh.add_external_action(e_hh_discharge, name="<
          Discharge>", beliefs=["ww_stored", "section"])
6     ab_be_hh.add_internal_action(i_hh_discharge, name="[
          Discharge]")
7
8     p_be_hh = PrimitiveTask(name="Be_Household",
9                             preconditions=Conditions(),
10                            effects=[],
11                            action_block=ab_be_hh)
12
13    m_be_hh = Method(conditions=Conditions(),
14                     subtasks=[p_be_hh],
15                     name="m_be_hh")
16
17    c_be_hh = CompoundTask(methods=[m_be_hh],
18                           name="c_be_hh")
19
20    beliefs = BeliefSet(ww_stored=Water(), ww_production=
          production, wwtp=wwtp, section=section, distance=
          distance)
21
22    behavior = Behavior()
23
24    planner = HTNPlanner(c_be_hh, verbose=True)
25
26    return beliefs, behavior, planner
```

Industry actions:

```
1  def i_ind_production(agent, beliefs):
2      beliefs["produced"] = True
3      if beliefs["max_production"] >\
4              (beliefs["storage_cap"] - beliefs["ww_stored"].
                   volume) / beliefs["ww_production"].volume:
5          produced_tons =  (beliefs["storage_cap"] - beliefs["
               ww_stored"].volume) / beliefs["ww_production"].
               volume
6          beliefs["storage_available"] = False
7      else:
8          produced_tons = beliefs["max_production"]
9          beliefs["storage_available"] = True
10
11     beliefs["total_production"] += produced_tons
12     beliefs["budget"] += produced_tons * beliefs["euros_per_ton
           "]
13     beliefs["ww_stored"] += (beliefs["ww_production"] *
           produced_tons)
```

```
14      return beliefs, False, "{0}\t{1}\t{2}\t{3}".format(beliefs[
            "budget"], produced_tons,
15                                                  (beliefs["
                                                        ww_production
                                                        "] *
                                                        produced_tons
                                                        ), beliefs[
                                                        "ww_stored"
                                                        ])
16
17
18  def i_ind_request_price(agent, beliefs):
19      beliefs["to_propose"] = beliefs["ww_stored"].copy()
20      beliefs["to_propose"].volume -= (beliefs["ww_proposed"] +
            beliefs["to_discharge"].volume)
21      beliefs["ww_proposed"] += beliefs["to_propose"].volume
22      return beliefs
23
24
25  def msg_ind_request_price(agent, beliefs, directory):
26      if beliefs["to_propose"].volume == 0:
27          return []
28      else:
29          return [(beliefs["wwtp"], "proposal", {"ww": beliefs["
                to_propose"], "distance": beliefs["distance"]},
                False)]
30
31
32  def i_ind_discharge(agent, beliefs):
33      beliefs["ww_stored"].volume -= beliefs["to_discharge"].
            volume
34      beliefs["storage_available"] = True
35      beliefs["budget"] -= beliefs["to_discharge"].volume *
            beliefs["price"]
36      beliefs["permission"] = False
37      return beliefs, False, "-{0}".format(beliefs["to_discharge"
            ])
38
39
40  def e_ind_discharge(environment, to_discharge=Water(), section
        =-1):
41      environment["sewers"][environment["sewer_id"][section]].
            sections[section].water += to_discharge
42      return environment
43
44
45  def msg_ind_report(agent, beliefs, directory):
46      return [(beliefs["wwtp"], "inform", {"ww": beliefs["
```

```
        to_discharge"], "distance": beliefs["distance"]}, False
        )]
```

Industry behavior:

```
class IndustryBehavior(Behavior):
    def perceive(self, environment, beliefs):
        beliefs["produced"] = False
        beliefs["to_discharge"].volume = 0.0
        return beliefs

    def process(self, message, beliefs, planner):
        if message.performative == "accept-proposal":
            beliefs["permission"] = True
            beliefs["price"] = message.content["price"]

            beliefs["ww_proposed"] -= message.content["volume"]
            if beliefs["price"] * message.content["volume"] <=
               beliefs["budget"]:
                beliefs["to_discharge"] = beliefs["ww_stored"].
                   copy()
                beliefs["to_discharge"].volume = message.
                   content["volume"]
            else:
                beliefs["to_discharge"] = beliefs["ww_stored"].
                   copy()
                beliefs["to_discharge"].volume = beliefs["
                   budget"] / beliefs["price"]
        elif message.performative == "reject-proposal":
            beliefs["permission"] = False
            beliefs["price"] = float("inf")
            beliefs["to_discharge"] = Water()
        return beliefs, planner, None
```

Industry HTN initialization:

```
def ind_init(max_production, storage_cap, ww_production,
    euros_per_ton, wwtp, section, distance):
    ab_production = ActionBlock()
    ab_production.add_internal_action(i_ind_production, "[
       Produce]")

    p_produce = PrimitiveTask(name="Produce",
                              preconditions=Conditions(),
                              effects=[Effect("ww_stored", "add
                                 ", Water(1.0))],
                              action_block=ab_production)

```

```
10      m_produce = Method(conditions=Conditions(storage_available=
            True),
11                         subtasks=[p_produce],
12                         name="m_produce")
13
14      c_production = CompoundTask(methods=[m_produce],
15                                  name="c_production")
16
17      ab_request = ActionBlock()
18      ab_request.add_internal_action(i_ind_request_price, "[
            Request Price]")
19      ab_request.add_message(msg_ind_request_price, "{Request
            Price}")
20
21      p_request = PrimitiveTask(name="Request Price",
22                                preconditions=Conditions(),
23                                effects=[],
24                                action_block=ab_request)
25
26      ab_discharge = ActionBlock()
27      ab_discharge.add_internal_action(i_ind_discharge, "[
            Discharge]")
28      ab_discharge.add_external_action(e_ind_discharge, "<
            Discharge>", ["to_discharge", "section"])
29      ab_discharge.add_message(msg_ind_report, "{Report}")
30
31      p_discharge = PrimitiveTask(name="Discharge",
32                                  preconditions=Conditions(
                                        permission=True),
33                                  effects=[Effect("
                                        storage_available", "rep",
                                        True)],
34                                  action_block=ab_discharge)
35
36      m_manage_stored = Method(conditions=Conditions(ww_stored
            =(0.01, float("inf")))),
37                               subtasks=[p_request, p_discharge],
38                               append=True,
39                               name="m_manage_stored")
40
41      c_manage_stored = CompoundTask(methods=[m_manage_stored],
42                                     name="c_manage_stored")
43
44      m_be_industy = Method(conditions=Conditions(),
45                            subtasks=[c_production,
                                  c_manage_stored],
46                            name="m_be_industry")
47
```

```
48      c_be_industry = CompoundTask(methods=[m_be_industy],
49                                   name="c_be_industry")
50
51      beliefs = BeliefSet(max_production=max_production,
           storage_cap=storage_cap, ww_stored=Water(),
52                          ww_production=ww_production,
                               total_production=0.0, budget=0.0,
                               euros_per_ton=euros_per_ton,
53                          wwtp=wwtp, section=section,
                               to_discharge=Water(), price=float("
                               inf"), permission=False,
54                          storage_available=True, distance=
                               distance, produced=False,
                               ww_proposed=0.0,
55                          to_propose=Water())
56
57      behavior = IndustryBehavior()
58
59      planner = HTNPlanner(c_be_industry,verbose=True)
60
61      return beliefs, behavior, planner
```

WWTP actions:

```
1  def i_wwtp_store(agent, beliefs):
2      beliefs["total_stored"] = True
3      if beliefs["incoming_ww"].volume < beliefs["
           entrance_storage_cap"] - beliefs["entrance_stored"].
           volume:
4          beliefs["entrance_stored"] += beliefs["incoming_ww"]
5          new_water = beliefs["incoming_ww"].copy()
6
7          beliefs["available_entrance_storage"] = beliefs["
               entrance_storage_cap"] - beliefs["entrance_stored"
               ].volume
8          beliefs["to_bypass"] = Water()
9      else:
10         beliefs["available_entrance_storage"] = 0.0
11         if beliefs["incoming_ww"].volume == beliefs["
               entrance_storage_cap"] - beliefs["entrance_stored"
               ].volume:
12             beliefs["entrance_stored"] += beliefs["incoming_ww"
                   ]
13             new_water = beliefs["incoming_ww"].copy()
14             beliefs["to_bypass"] = Water()
15         else:
16             new_water = beliefs["incoming_ww"].copy()
```

```
17              new_water.volume = beliefs["entrance_storage_cap"]
                    - beliefs["entrance_stored"].volume
18              beliefs["entrance_stored"] += new_water
19              beliefs["incoming_ww"].volume -= new_water.volume
20              beliefs["to_bypass"] = beliefs["incoming_ww"]
21      beliefs["incoming_ww"] = Water()
22      return beliefs, False, "+{0}\t-{1}".format(new_water,
            beliefs["to_bypass"])
23
24
25  def e_wwtp_bypass(environment, to_bypass=Water(), out_section
        =-1):
26      environment["river"].sections[out_section].water +=
            to_bypass
27      return environment
28
29
30  def i_wwtp_bypass(agent, beliefs):
31      beliefs["to_bypass"] = beliefs["incoming_ww"]
32      beliefs["incoming_ww"].volume = 0.0
33      return beliefs
34
35
36  def i_wwtp_treat(agent, beliefs):
37      for t in beliefs["treatment"]:
38          t["duration"] += 1
39
40      if beliefs["treatment"] and beliefs["treatment"][-1]["
            duration"] >= beliefs["treatment_time"]:
41          t = beliefs["treatment"].pop()
42          beliefs["treatment_stored"] -= t["ww"].volume
43          t["ww"].ss *= beliefs["treatment_effect"]["ss"]
44          t["ww"].bod *= beliefs["treatment_effect"]["bod"]
45          t["ww"].cod *= beliefs["treatment_effect"]["cod"]
46          t["ww"].tn *= beliefs["treatment_effect"]["tn"]
47          t["ww"].tp *= beliefs["treatment_effect"]["tp"]
48          beliefs["to_discharge"] += t["ww"]
49
50      if beliefs["entrance_stored"].volume > 0:
51          if beliefs["treatment_cap"] - beliefs["treatment_stored
                "] >= beliefs["entrance_stored"].volume:
52              beliefs["treatment"].insert(0, {"ww": beliefs["
                    entrance_stored"].copy(), "duration": 0})
53              beliefs["treatment_stored"] += beliefs["
                    entrance_stored"].volume
54              beliefs["entrance_stored"].volume = 0.0
55              beliefs["available_entrance_storage"] = beliefs["
                    entrance_storage_cap"]
```

```
56          else:
57              new_water = beliefs["entrance_stored"].copy()
58              new_water.volume = beliefs["treatment_cap"] -
                    beliefs["treatment_stored"]
59              beliefs["treatment"].insert(0, {"ww": new_water, "
                    duration": 0})
60              beliefs["treatment_stored"] += new_water.volume
61              beliefs["entrance_stored"].volume -= new_water.
                    volume
62              beliefs["available_entrance_storage"] = beliefs["
                    entrance_storage_cap"] - beliefs["
                    entrance_stored"].volume
63
64      return beliefs, False, "{0}".format(beliefs["treatment"])
65
66
67  def e_wwtp_discharge(environment, to_discharge=Water(),
        out_section=-1):
68      environment["river"].sections[out_section].water +=
            to_discharge
69      return environment
70
71
72  def i_wwtp_discharge(agent, beliefs):
73      if beliefs["entrance_stored"] == 0 and beliefs["
            treatment_stored"] == 0:
74          beliefs["total_stored"] = False
75
76      return beliefs
```

WWTP behavior:

```
1   class WWTPBehavior(Behavior):
2       def perceive(self, environment, beliefs):
3           beliefs["to_discharge"].volume = 0.0
4           if environment["sewers"][environment["sewer_id"][
                beliefs["in_section"]]].sections[beliefs["
                in_section"]].water.volume > 0:
5               beliefs["incoming_ww"] = \
6                   environment["sewers"][environment["sewer_id"][
                        beliefs["in_section"]]].sections[beliefs["
                        in_section"]].water
7           for i in range(len(beliefs["expected_ww"]) - 1):
8               beliefs["expected_ww"][i] = beliefs["expected_ww"][
                    i+1]
9
10          return beliefs
11
```

```
12      def process(self, message, beliefs, planner):
13          if message.performative == "inform":
14              beliefs["expected_ww"][message.content["distance"]
                    - 1] += message.content["ww"].volume
15              reply=None
16          elif message.performative == "proposal":
17              if beliefs["expected_ww"][message.content["distance
                    "]] + message.content["ww"].volume >=\
18                       (beliefs["entrance_storage_cap"] - beliefs[
                            "reserve"]):
19                  reply = message.reply("reject-proposal", None)
20              else:
21                  current = beliefs["expected_ww"][message.
                        content["distance"]] / beliefs["
                        entrance_storage_cap"]
22                  total = (beliefs["expected_ww"][message.content
                        ["distance"]] + message.content["ww"].
                        volume) \
23                           / beliefs["entrance_storage_cap"]
24                  G = (0.5 * total * total - 0.5 * current *
                        current + total - current) / 2 + 1
25                  price = 0
26                  price += message.content["ww"].ss * beliefs["
                        treatment_cost"]["ss"] / 1000.0
27                  price += message.content["ww"].bod * beliefs["
                        treatment_cost"]["bod"] / 1000.0
28                  price += message.content["ww"].cod * beliefs["
                        treatment_cost"]["cod"] / 1000.0
29                  price += message.content["ww"].tn * beliefs["
                        treatment_cost"]["tn"] / 1000.0
30                  price += message.content["ww"].tp * beliefs["
                        treatment_cost"]["tp"] / 1000.0
31                  price *= G
32                  reply = message.reply("accept-proposal", {"
                        price": price, "volume": message.content["
                        ww"].volume})
33          else:
34              reply = None
35
36          return beliefs, planner, reply
```

WWTP HTN initialization:

```
1 def wwtp_init(entrance_storage_cap, treatment_time,
      treatment_effect, treatment_cap, treatment_cost,
2                 in_section, out_section, reserve, max_distance):
3     ab_store = ActionBlock()
4     ab_store.add_internal_action(i_wwtp_store, "[Store]")
```

```
5     ab_store.add_external_action(e_wwtp_bypass, "<Bypass>", ["
          to_bypass", "out_section"])
6
7     p_store = PrimitiveTask(name="Store",
8                             preconditions=Conditions(),
9                             effects=[Effect("total_stored", "
                                  rep", True)],
10                            action_block=ab_store)
11
12    m_store = Method(conditions=Conditions(incoming_ww=(0.01,
          float("inf")),
13                                           available_entrance_storage
                                                 =(0.01, float("
                                                 inf"))),
14                     subtasks=[p_store],
15                     name="m_store")
16
17
18    ab_bypass = ActionBlock()
19    ab_bypass.add_internal_action(i_wwtp_bypass, "[Bypass]")
20    ab_bypass.add_external_action(e_wwtp_bypass, "<Bypass>", ["
          to_bypass", "out_section"])
21
22    p_bypass = PrimitiveTask(name="Bypass",
23                             preconditions=Conditions(),
24                             effects=[],
25                             action_block=ab_bypass)
26
27    m_bypass = Method(conditions=Conditions(incoming_ww=(0.01,
          float("inf"))),
28                      subtasks=[p_bypass],
29                      name="m_bypass")
30
31    c_incoming_ww = CompoundTask(methods=[m_store, m_bypass],
32                                 name="c_incoming_ww")
33
34
35    ab_treat = ActionBlock()
36    ab_treat.add_internal_action(i_wwtp_treat, "[Treat]")
37    ab_treat.add_external_action(e_wwtp_discharge, "<Discharge>
          ", ["to_discharge", "out_section"])
38    ab_treat.add_internal_action(i_wwtp_discharge, "[Discharge]
          ")
39
40    p_treat = PrimitiveTask(name="Treat",
41                            preconditions=Conditions(),
42                            effects=[],
43                            action_block=ab_treat)
```

```
44
45      m_treat = Method(conditions=Conditions(total_stored=True),
46                       subtasks=[p_treat],
47                       name="m_treat")
48
49      p_idle = PrimitiveTask(name="Idle",
50                             preconditions=Conditions(),
51                             effects=[],
52                             action_block=ActionBlock())
53
54      m_idle = Method(conditions=Conditions(),
55                      subtasks=[p_idle],
56                      name="m_treat")
57
58      c_stored_ww = CompoundTask(methods=[m_treat, m_idle],
59                                 name="c_stored_ww")
60
61
62      m_be_wwtp = Method(conditions=Conditions(),
63                         subtasks=[c_incoming_ww, c_stored_ww],
64                         append=True,
65                         name="m_be_wwtp")
66
67      c_be_wwtp = CompoundTask(methods=[m_be_wwtp],
68                               name="c_be_wwtp")
69
70
71      beliefs = BeliefSet(incoming_ww=Water(),
           entrance_storage_cap=entrance_storage_cap,
           entrance_stored=Water(),
72                          available_entrance_storage=
                               entrance_storage_cap, to_bypass=
                               Water(), in_section=in_section,
73                          out_section=out_section, treatment=[],
                               reserve=reserve, treatment_time=
                               treatment_time,
74                          treatment_effect=treatment_effect,
                               to_discharge=Water(), treatment_cap
                               =treatment_cap,
75                          treatment_stored=0.0, messages=[],
                               expected_ww=[0.0 for d in range(
                               max_distance + 1)],
76                          treatment_cost=treatment_cost,
                               total_stored=False)
77
78      behavior = WWTPBehavior()
79
80      planner = HTNPlanner(c_be_wwtp, verbose=True)
```

```
81
82      return beliefs, behavior, planner
```

Poststep function:

```
1  def poststep(environment):
2      environment["river"].flow()
3      for sewer in environment["sewers"]:
4          sewer.flow()
5
6      return environment
```